Fast, Realistic Terrain Synthesis

Thesis by

Justin Crause

In fulfilment of the requirements
for the degree of
Master of Science

Supervised by

Assoc. Prof. James Gain

Assoc. Prof. Patrick Marais

July 2015

Plagiarism Declaration

I know the meaning of Plagiarism and declare that all of the work in the document, save for that which is properly acknowledged, is my own.

Abstract

The authoring of realistic terrain models is necessary to generate immersive virtual environments for computer games and film visual effects. However, creating these landscapes is difficult – it usually involves an artist spending many hours sculpting a model in a 3D design program. Specialised terrain generation programs exist to rapidly create artificial terrains, such as Bryce (2013) and Terragen (2013). These make use of complex algorithms to pseudo-randomly generate the terrains, which can then be exported into a 3D editing program for fine tuning. Height-maps are a 2D data-structure, which stores elevation values, and can be used to represent terrain data. They are also a common format used with terrain generation and editing systems. Height-maps share the same storage design as image files, as such they can be viewed like any picture and image transformation algorithms can be applied to them.

Early techniques for generating terrains include fractal generation and physical simulation. These methods proved difficult to use as the algorithms were manipulated with a set of parameters. However, the outcome from changing the values is not known, which results in the user changing values over several iterations to produce their desired terrain. An improved technique brings in a higher degree of user control as well as improved realism, known as texture-based terrain synthesis. This borrows techniques from texture synthesis, which is the process of algorithmically generating a larger image from a smaller sample image. Texture-based terrain synthesis makes use or real-world terrain data to produce highly realistic landscapes, which improves upon previous techniques. Recent work in texture-based synthesis has focused on improving both the realism and user control, through the use of sketching interfaces.

We present a patch-based terrain synthesis system that utilises a user sketch to control the location of desired terrain features, such as ridges and valleys. Digital Elevation Models (DEMs) of real landscapes are used as exemplars, from which candidate patches of data are extracted and matched against the user's sketch. The best candidates are merged seamlessly into the final terrain. Because real landscapes are used the resulting terrain appears highly realistic. Our research contributes a new version of this approach that employs multiple input terrains and acceleration using a modern Graphics Processing Unit (GPU). The use of multiple inputs increases the candidate pool of patches and thus the system is capable of producing more varied terrains. This addresses the limitation where supplying the wrong type of input terrain would fail to synthesise anything useful, for example supplying the system with a mountainous DEM and expecting deep valleys in the output. We developed a hybrid multithreaded CPU and GPU implementation that achieves a 45 times speedup.

Acknowledgements

Completing this task has been both a pleasure and a curse. I started out full of energy, enthusiasm and happy to explore this exciting work. But the sense of relaxed freedom in the 'master's environment' was to be short lived. I quickly realised the enormity of the task ahead and the time ticked by alarmingly. My progress slowed and fell below expectation and then I was tempted into the real world and the prospect of earning a proper living. My biggest challenge was explaining this to my supervisors, but I mustered up the courage and faced the difficult conversation...that was two years ago. Since then I've slowly but steadily plodded along, finding it challenging to balance a professional career with my studying and still have a social life. It has taken me longer than planned but I've had fantastic support and guidance from my supervisors James and Patrick, who never wavered in their support of my efforts over the years. I have now reached the end of this road and completed my mammoth project. My heartfelt thanks to all my friends and family who have encouraged and motivated me, even when I was thinking of throwing in the towel!

Looking back on it now, it was an amazing adventure. There will be many memories of long days in the lab with friends, the morning muffins from our weekly meetings and the pub lunches afterwards – celebrating the end of yet another week. I am now able to close this chapter of life and start work on my next adventure.

To everyone that made this possible, THANK YOU.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Detailed terrain models are a fundamental component of many 3D scenes used in computer games (Figure 1.1) and the creation of film visual effects (Figure 1.2). The creation of realistic artificially-generated terrain helps the gamer or audience feel immersed in the environment. In some instances, where the landscape is only used as a visual backdrop with no user interaction, a simple two-dimensional (2D) terrain profile might be satisfactory. This profile can be either a hand drawn graphic or an image of a real landscape. This technique was used in early games and virtual environments to reduce the space requirements and computational complexity. However, it is more often a requirement that the environment be navigable, which requires a three-dimensional (3D) landscape. Creating these landscapes is no easy task – it usually involves an artist spending many hours tweaking a 3D mesh structure. As the requirements for larger, more realistic and detailed terrains increase so does the complexity and amount of time required to manually create them. As an alternative artists can make use of real landscapes in the form of digital elevation models (DEMs) that can be obtained from the US Geological Survey (USGS, 2013). These provide true realism but often do not match up with the artist's vision, thus requiring manual editing. This has led to great interest in the procedural generation of terrain models. Procedural methods are algorithms that allow for the quick generation of data with little user input.

Figure 1.1: Example of a landscape generated for an upcoming game The Witcher III (2015)

Terrain synthesis is the process of creating an artificial landscape algorithmically using procedural methods. The two most common procedural methods are fractal generation and physical simulation. These generate terrains with a minimal amount of user input in the form of algorithm parameters. These parameters are usually unintuitive and many iterations of synthesis may be required before an optimal set of parameters is found to generate a suitable terrain. Software packages such as Bryce (2013) and Terragen (2013) can be used, but in most cases the artist will still need to tweak

the terrain to achieve the desired look. These packages use methods that pseudo-randomly displace height values of an initially flat terrain model according to a given fractal technique. Furthermore, these programs are unable to simulate physical weathering patterns, and generated terrain models must be exported to some other system to add such detail. An erosion system will enhance the realism of the input terrain but requires the user to have a fair understanding of erosion models and is also computationally expensive. These programs can more rapidly generate terrains but the results are somewhat random. A system that allows the user to specify terrain constraints and produces a realistic-looking terrain that closely matches the user's expectations would be ideal.

Figure 1.2: Still from the movie Avatar (2009) with computer generated landscape.

An alternative procedural method is example-based, which works by utilising existing terrain data, often in the form of Digital Elevation Models (DEMs) commonly from the USGS (2013), and recombining them using texture synthesis techniques. Current state-of-the-art systems using this method are those by Zhou et al. (2007) and Tasse et al. (2011). The user specifies their requirements in the form of a sketch, which provides the location of certain dominant features, such as mountains and valleys. The system then takes this sketch and breaks it up into small blocks or patches which it then searches for the best match from a pool of candidates – patches taken from the DEM files with feature rich characteristics. For the areas where no features are described, the system will populate the terrain with insignificant data – candidates with no dominant feature characteristics. The use of DEMs as the input source produces terrains that appear highly realistic. Combining realism with the flexibility of a sketching interface provides a good system for synthesising artificial terrains. There are some issues with the current implementations, which include being slow to execute and limited with the variability of the terrain when using only a single input source. These are two key areas for improvement.

1.1 Aims

The primary objective of this research thesis is to build a terrain synthesis system to rapidly generate realistic terrains from the input of a simplified user interface. The system builds on previous work by Tasse et al. (2011) and provides several extensions to improve the synthesis results. To facilitate this objective, the following key requirements were identified:

- A system capable of producing realistic terrains, making use of landscape data from the United States Geological Survey (2013). A user study conducted by Tasse et al. (2011) confirmed that their system produces terrain that is more realistic than ones generated by a multi-resolution deformation (a procedural synthesis method). The same techniques will be incorporated into this research with the results being compared to the system by Tasse et al. (2011).

- A simple interface that allows the user to sketch out the placement of both ridge and valley line features to describe the overall design of their terrain.

- Make use of a large collection of input terrains to increase the candidate pool for synthesis. When using a single input terrain the variability of features is constrained by the amount of sample data available. Using multiple input sources allows for better quality, more diverse terrains. This objective represents the novel contribution of this research.

- Accelerate the process by implementing CPU caching algorithms and optimising the process to reduce the synthesis time.

- Further accelerating the synthesis process with the aid of programmable Graphics Processing Units (GPUs) and NVIDIA's Compute Unified Device Architecture (CUDA). Modern GPUs have become more powerful than CPUs by orders of magnitude for certain computations that can be parallelised, such as scientific data processing. CUDA is an application interface developed to enable General Purpose GPU (GPGPU) computing. This has spawned a new era in computational research focusing on parallel computation. Texture synthesis is one field which benefits from parallel computation. We make use of this to dramatically reduce the time it takes to complete a synthesis option, thus making our system suitable as a rapid prototyping tool. We combine the CPU & GPU optimisations to create a hybrid system for maximum performance.

- The system will be evaluated with visual inspection to verify that the realism of our output matches the quality of the previous system. Speedup comparisons will be made between all the different CPU and GPU versions to evaluate their performance.

1.2 Contributions

The main contribution for this research is the introduction of multiple input sources to increase the variety of data available during synthesis operations. Prior work with patch-based systems focuses on the use of a single input source to synthesise the terrains (Zhou et al., 2007, Tasse et al., 2011). This is reliant on the user selecting the correct source to get the best results as some sources might not contain the correct features required. We show that our system is capable of producing very large terrains, varied terrains. Our hybrid CPU-GPU implementation is capable of a 40 times speedup over a single-core CPU system.

1.3 Thesis structure

The structure of the thesis is as follows:

- Chapters 2 and 3 contain background information on procedural terrain generation and Graphics Processing Units (GPUs) respectively. GPUs can be used to accelerate computation of parallel algorithms and we use them to reduce synthesis times of our system.

- Chapter 4 provides a detailed analysis of the system developed by Tasse et al. (2011), which we extend in this thesis. The limitations of this system are highlighted together with our proposed improvements.

- Chapter 5 presents the overview of our system, focusing on our new contributions to example-based terrain generation.

- Chapters 6 and 7 describe the core components of feature and non-feature synthesis in detail. This includes the various CPU and GPU versions we developed while improving and optimising the system.

- Results are presented in Chapter 8 which compares our new system to that of Tasse et al. (2011). A single core CPU implementation is compared to a hybrid approach, which incorporates multiple threads and a GPU to accelerate the synthesis stage. Visual assessment is used to verify that realism is preserved with our proposed modifications.

- Chapter 9 concludes the thesis and lists some possible avenues of future work to improve and further accelerate the synthesis of terrains.

2 Background: Terrain Generation

This chapter provides an overview of methods to procedurally generate terrains. We begin by describing common representations of terrain data (Section 2.1) and follow this with a discussion of important generation techniques (Section 2.2). A summary of the techniques and motivation for our choice of synthesis concludes this chapter.

2.1 Terrain Representation

The simplest representation of terrains is as a two-dimensional grid-based data-structure. This data-structure is commonly represented as an image known as a *height-map*. Height-maps are easy to use given their uniform grid-based nature, where each entry stores a height value for the corresponding location on the terrain. Figure 2.1 shows a simple example of a height-map represented as a 2D image (shown left) and the corresponding 3D rendering on the right. The pixel's intensity represents the height of the terrain and is stored in a single channel of the image, resulting in a grayscale image. The USGS (United States Geological Survey) have surveyed many real landscapes and made available in a height-map digital form commonly referred to as *Digital Elevation Models* (DEMs). These DEMs are freely available from the USGS website (USGS, 2013). Height-maps can be encoded using a variable number of bits. If only a single channel (8-bit) is used, this allows only 255 possible height values, which is insufficient for replicating highly detailed terrain. The number of bits used depends largely on the format, with most DEM files being stored using 16-bit images, giving 65,535 height values.

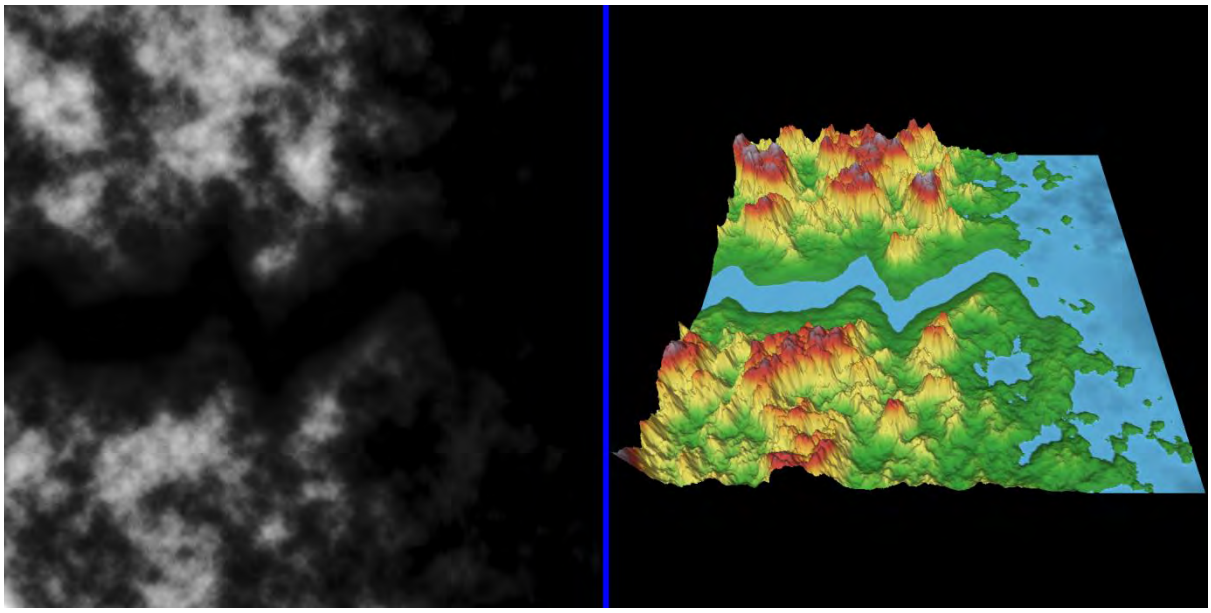


Figure 2.1: Example of height-map. 2D image shown on left with corresponding 3D rendering on the right. Generated and rendered in GeoGen (2013)

The regular grid structure of height-maps facilitates storage efficiency and ease of implementation and is well suited to filter-based image processing. However, height-maps are not without limitations. For instance, they lack the ability to represent overhangs, caves or structures where a given location needs multiple height values.

Terrain models can also be represented as a mesh of polygons, usually triangles. *Triangulated Irregular Networks* (TINs) are a type of mesh structure in which the terrain is composed of a set of connected, variably sized triangles (Peucker et al., 1978). The triangles vertices are adaptively chosen, often with a Delaunay triangulation algorithm (Fowler and Little, 1979), to produce an accurately representation of the terrain. TINs are able to capture three-dimensional structures such as caves, where a height-map would fail, and also support a level-of-detail (LOD) system: higher density areas are represented with many small, tightly-packed triangles and smoother, less detailed areas with fewer larger triangles. As a result of the LOD system, the storage overhead for TINs is small; they are, however, more difficult to manipulate procedurally due to their non-uniform structure. An example of a TIN model is provided in Figure 2.2. TINs are more appropriate for manual terrain modelling or rendering systems as these packages are designed to work with vertices at non-uniform locations. For more details on TINs we refer the reader to Abdelguerfi et al. (1998) and Pajarola et al. (2002).

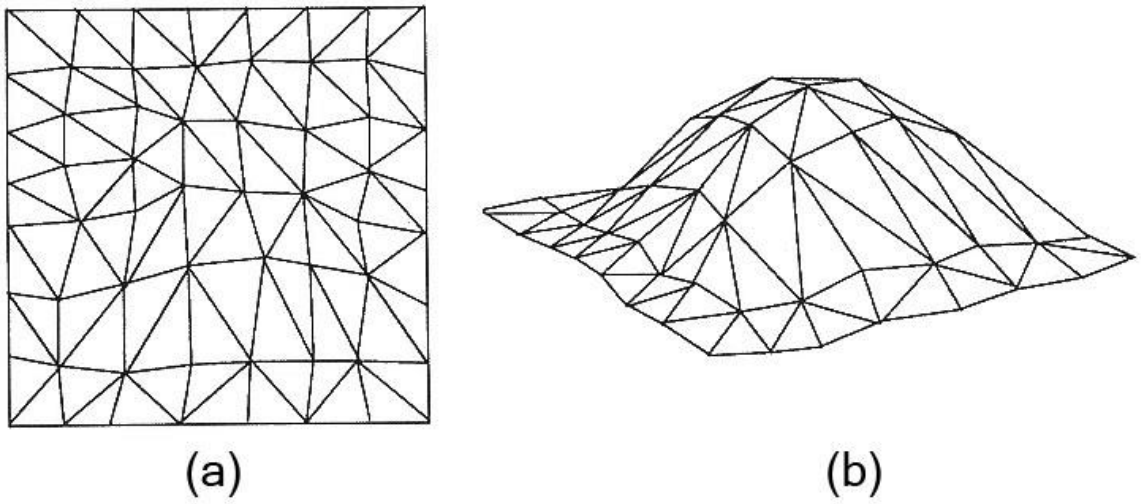


Figure 2.2: Triangulated Irregular Network format. (a) Top-down representation. (b) Perspective view

Voxels (volumetric elements) are another way of representing terrains (Kaufman et al., 1993, Dorsey et al., 1999). Voxels are the 3D equivalent to a 2D pixel. They are aligned in a three-dimensional grid structure with their locations inferred from their index in the grid. Voxels can store data such as colour and opacity, which together create 3D structures. As such voxel-grids are capable of producing terrains with caves and other 3D structures. They are also widely used in the scientific and medical domains. However, they have a large memory and storage overhead. This impacts on rendering performance and restricts the size of structure that can be represented. A good example of a voxel-based environment is from the popular video game Minecraft (2015), as seen in Figure 2.3.

Another example of representing volumetric data is through using a system of particles, to simulate granular materials such as sand. Bell et al. (2005) present such a system with non-spherical particles. Granular materials behave differently compared to fluids because they can flow down a slope like fluid and they can also form a static volume like a solid. These systems are more suited for small-scale simulations where dynamic interactions are required as they require complex algorithms to simulate the inter-particle interactions. Longmore et al. (2013) extend this work to leverage the

parallel processing capabilities of modern GPUs. However, while more efficient than a CPU-based implementation, the system is only intended for small-scale volumes due to it being computationally expensive. The system uses 3D textures to store the particle information, which requires a large amount of memory and limits the number of particles that can be simulated. These limitations prevent us from utilising particles to represent a large terrain.

Figure 2.3: Screengrab of the generated landscape in Minecraft

Height-maps are the format most widely supported by common terrain generation packages (Terragen, 2013, Bryce, 2013, WorldMachine, 2013). These packages make use of image processing functions, which are easy to implement on height-map images. Another reason to use height-maps is that real landscape data produced from aerial or satellite surveys is stored in this format. Since our research will make extensive use of DEM images, and extends an existing height-map based approach, our synthesis system is also based on height-map data-structures.

2.2 Terrain Generation

Terrain generation is the process of creating an artificial landscape using procedural algorithmic methods. Artificial terrains have many applications, including virtual environments, computer games and movies. Terrains can be manually sculpted in 3D design programs but this is time consuming. Fortunately, the process can be accelerated through the use of procedural methods. There are three broad categories of procedural terrain generation techniques: *Fractal*, *Physics* and *Texture-based*. A fractal surface is generated using a stochastic algorithm designed to produce fractal behaviour that mimics that of a natural landscape. Physical simulations generally enhance the realism of a fractal surface by applying erosion techniques to the surface. Finally, texture-based methods borrow techniques from texture synthesis and typically copy data from a source image to build a new terrain. Specialised programs such as Terragen (2013) and Bryce (2013) incorporate a number of procedural methods for generating terrains quickly. However, these implementations only use

fractal techniques and may allow for erosion. We will show that in many cases such an approach is not suitable when the user has a specific terrain design in mind. Each category is described in the subsections below.

2.2.1 Fractal-based generation

Fractal methods were introduced by Benoit Mandelbrot in his seminal book, "*The Fractal Geometry of Nature*" (Mandelbrot, 1983). He observed that natural shapes often contain self-similar patterns: magnified areas are statistically similar to the original shape. He introduced Fractal Geometry, which is a mathematical representation for natural shapes that are not easily described by Euclidean geometry. The term 'fractal-based' has been applied more loosely over the years and as such not all the techniques discussed in this section are truly fractal. Here the term classifies techniques that generate terrains that exhibit self-similar patterns even if the algorithm is not mathematically fractal.

Fractional Brownian motion (fBm) describes the process of representing these self-similar shapes. It is also known as the "Random Walk Process" and consists of a series of steps in a random direction, where the steps are normally distributed with a mean of zero and variance representing the roughness. In terms of terrain generation, this involves a series of iterations of a stochastic algorithm. Mandelbrot reasoned that if this process were extended in two dimensions the resulting "Brownian surface" could be a visual approximation of a landscape in nature. Some of his work explored the creation of these types of surfaces (Mandelbrot, 1975). Numerous researchers have extrapolated Mandelbrot's research and adapted it to produce fractal-based terrains (Fournier et al., 1982, Voss, 1985, Miller, 1986, Lewis, 1987, Musgrave et al., 1989, Saupe, 2003). One of the earliest known images of a Brownian Surface is presented in Figure 2.4; it is part of a sequence of fractional Brown Islands.

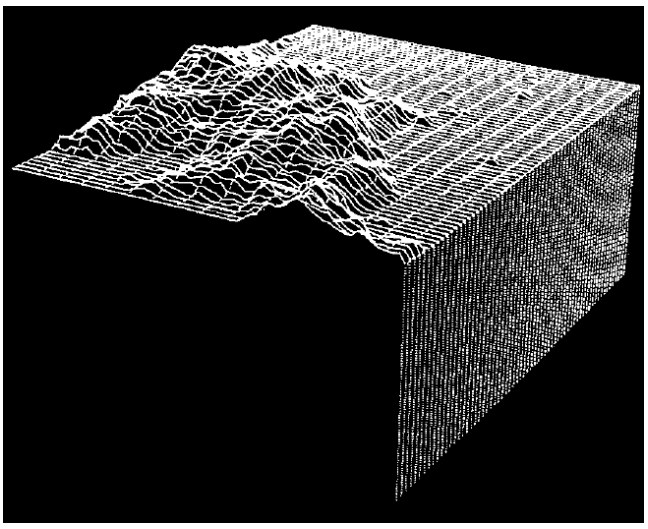


Figure 2.4: One of the earliest known examples of a Brownian Surface: Fractal Brown Islands (Mandelbrot, 1983)

Poisson Faulting is one of the earliest forms of fractal terrain generation (Mandelbrot, 1983, Voss, 1985). This technique involves applying a series of Gaussian random displacements (faults) to a plane. In simpler terms, a line is chosen across the plane and one side displaced by a random height. This height value is reduced after each fault to avoid abrupt height changes in the final resulting terrain. Figure 2.5 shows an example of the faulting process, captured at various synthesis stages.

This was employed by Mandelbrot to create fractal coastlines (Mandelbrot, 1975) and fractal planets by Voss (1985). Faulting has a fixed resolution, which means there is no consideration of level-of-detail (LOD). LOD is important in terrains as features are present on different scales, such as large scale mountains at a coarse level and cracks on a fine level. These techniques also suffer from an $O(n^3)$ runtime, depending on the resolution and number of iterations, which severely impacts performance. This led to the development of subdivision methods, discussed next.

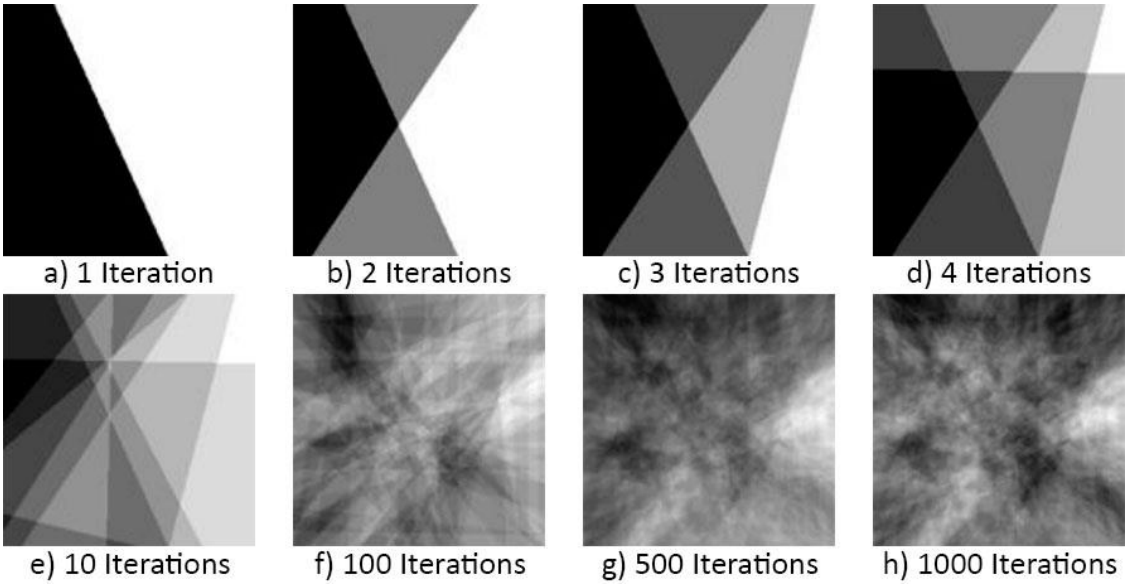


Figure 2.5: Example of Poisson Faulting over several iterations

Subdivision methods work by iteratively adding finer levels of detail by dividing the current terrain level. Midpoint-displacement is an example of this and is used to generate terrains (Fournier et al., 1982, Miller, 1986, Lewis, 1987, Mandelbrot, 1988, Saupe, 2003). There a many midpoint-displacement techniques, usually differing in the way points are interpolated during each step. A simple example starts with a quad on a plane and randomly assigns the corners with seeding values. This quad is then divided into four smaller quads. The values of the corners of the new quads are interpolated between the corners of the parent quad. The midpoint value is additionally offset by a random value controlled by the desired roughness of the terrain. This process is repeated on each of the new quads until a desired LOD is obtained. This is shown in Figure 2.6; the simple process outlined above is easily implemented and can run in linear time. Many subdivision methods are subject to the "creasing problem" (Miller, 1986). This is the occurrence of creases or slopes along the quad boundaries which are visually noticeable. A possible solution to this is to also apply a displacement to all the points, not just the midpoint, this is called "successive random addition" (Saupe, 1989). This leads to a significant amount of additional calculations being required, leading to a general preference for simple Midpoint-displacement. Both these methods produce unnatural repeating patterns in the terrain. Furthermore, only a single parameter is available to control terrain roughness, which limits user control. Nonetheless, due to its fast generation time, most terrain generation packages (such as Terragen (2013)).

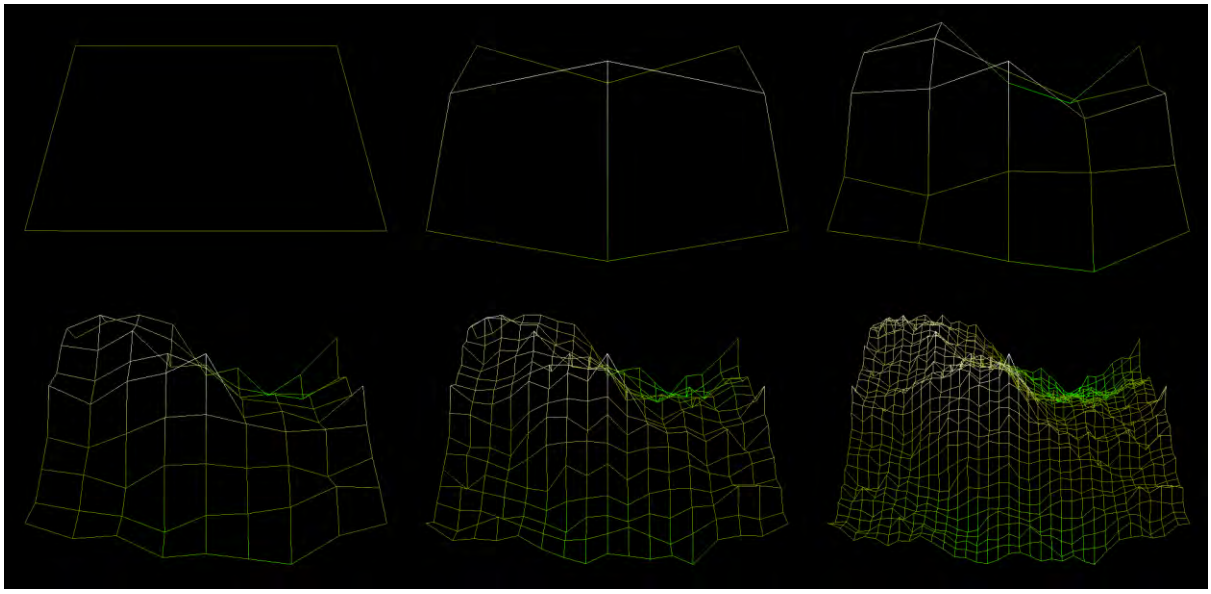


Figure 2.6: The first 6 iterations of a Midpoint-Displacement algorithm

Procedural Noise Synthesis can be informally defined as being the random number generator of computer graphics. It is random and unstructured in pattern and is used when there is a need for a source with extensive detail but lacking in evident structure. An example of a terrain synthesised utilising noise synthesis is shown in Figure 2.7. These are popular methods used by commercial packages, such as Bryce (2013), and have been widely researched (Fournier et al., 1982, Saupe, 1991, Schneider et al., 2006). Terrains are generated with simple implementations that involve the summation of successive down-scaled copies of a band-limited noise function. This type of noise generating function was first introduced by Ken Perlin (1985) and has been improved over the years (Perlin, 2002). Typically each new copy contains a higher band-limited frequency with lower amplitude such that large scale features are generated in early iterations and finer detail in the later ones. A known problem with Perlin noise is that it is weakly band-limited as each band contains only frequencies in a power-of-two (Lewis, 1989). This leads to aliasing and loss of detail. *Wavelet noise* (Cook and DeRose, 2005) address these issues by taking an image filled with random noise (R) and downsampling to half its size ($R^{\downarrow}$). This image is then upsampled to full size ($R^{\downarrow\uparrow}$) and subtracted from the original image (R). This results in band-limited data that can be used in terrain generation. This method has the benefit of being almost perfectly band-limited and provides effective level of detail without the aliasing issues of Perlin noise (Cook and DeRose, 2005). Wavelet noise is also fast and easy to implement, leading to its use in terrain generation applications (de Carpentier, 2007, Gain et al., 2009, Cui, 2011). For more details, we refer interested readers to the survey of procedural noise functions by Lagae et al. (2010)

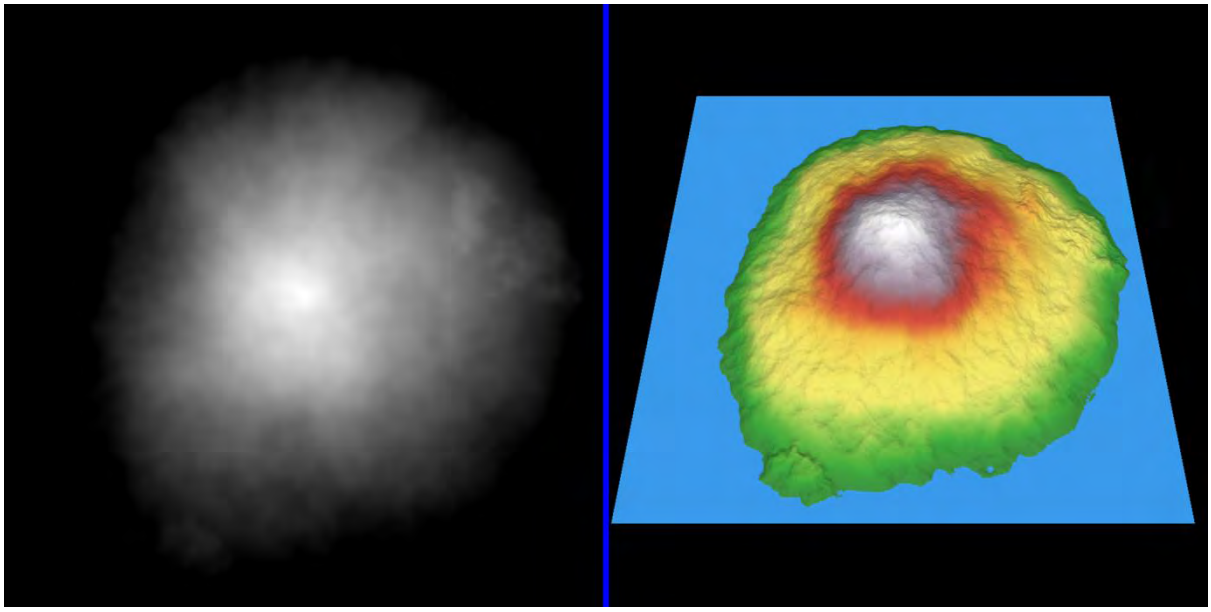


Figure 2.7: Example of terrain generated through noise synthesis. Generated and rendered in GeoGen (2013)

For additional information on fractal-based generation, we refer the reader to Ebert et al. (2003). Musgrave covers many fractal generation methods in his work "Methods for Realistic Landscape Imaging" (Musgrave, 1993). Fractal methods are easy to implement and widely supported by terrain generation programs. User control suffers as their parameters are not intuitive and one cannot easily generate specific variations of terrain. The generated terrain also lacks realism since it is missing structures that arise from natural weathering and erosion on landscapes. The generation of such aspects of realistic-looking terrains can be achieved by physics simulations; we provide a discussion of such techniques below.

2.2.2 Physics-based generation

Physics-based methods aim to improve the realism of artificially generated terrain by simulating the effects of erosion. Kelley et al. (1988) use hydrology data to generate stream network drainage patterns that can be used to determine the topography of a terrain surface. However, this method, while efficient, lacks the detail of a fractal surface since the terrain is modelled from the stream network. While the stream network controlling the generation may be fractal, the surface used as the initial terrain is not and cannot be made so without disturbing the drainage basins and stream paths. Musgrave et al. (1989) combine a fractal height-map with a hydraulic model. Water is dropped at each vertex and allowed to run off the terrain. The water erodes the surface by depositing material at different locations based on a sediment load function for the water passing over the vertex. Musgrave et al. (1989) also introduce a global model for simulation they based on thermal weathering.

Thermal weathering is a simulation where sharp changes in elevation are diminished by knocking material loose from steep inclines to eventually pile up at the bottom of the slope. This process iterates until the maximum angle of stability for the material (talus angle) is reached. This technique is simple to implement and runs efficiently (Musgrave et al., 1989, Marák et al., 1997, Olsen, 2004). The following equation is evaluated at every vertex to determine the movement of material:

$$h_{t+1}^{u} = \begin{cases} h_t^u + c_t(h_t^v - h_t^u - T), & h_t^v - h_t^u > T \\ h_t^u, & h_t^v - h_t^u \leq T \end{cases}$$

The difference in the height h of the current vertex v and its neighbour u is compared with the globally defined talus angle T. If its slope is greater than the talus angle then a fixed percentage $c_t$of the difference is moved onto the neighbour. Beneš and Forsbach (2001) introduce a terrain structure that is more suitable for realistic erosion algorithms. Their model consists of a 2D grid similar to a height-map but with each location storing an array representing different layers. Each layer stores information, such as the elevation, and material properties, such as density. This is a trade-off between a height-map and full voxel terrain. The model allows for air layers and, as such, cave structures can be created. Thermal erosion is suitable for deposition of material to smooth out steep slopes but does not simulate drainage patterns. This is achieved with hydraulic erosion.

Hydraulic (Fluvial) erosion is a simulation that deposits water at the vertices of the terrain and allows it to flow downhill, eroding the surface as it goes. This method has been extensively investigated in the literature (Kelley et al., 1988, D'Ambrosio et al., 2001, Beneš et al., 2006, Krištof et al., 2009). Hydraulic erosion is more complex than thermal erosion but is simply described by associating each vertex v at time t with a height h_t^v, volume of water w_t^v and an amount of sediment s_t^v, suspended in it. At each time step excess water and sediment is passed to the vertex's neighbours. There are two approaches to calculating hydraulic erosion; Eulerian and Lagrangian. Eulerian focuses on a fixed window and observes the particles affects only while they are in it. Whereas Lagrangian focuses on an individual particle and tracks its movement throughout the system (Krištof et al., 2009). Figure 2.8 shows an example of a terrain before and after hydraulic erosion has been applied. Nagashima (1998) use a 2D fractal river network to which thermal and hydraulic erosion simulations are applied, which erodes the banks. Beneš and Forsbach (2001) improve on previous work by distributing sediment to the vertex's eight neighbours and implementing evaporation to simulate water pools drying up.

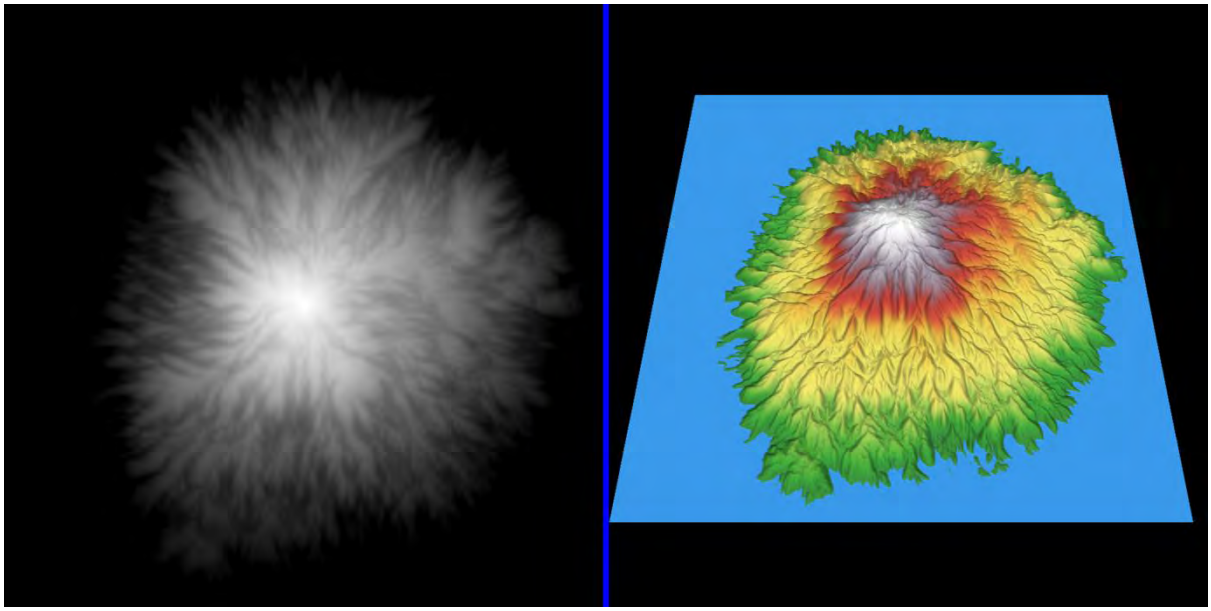


Figure 2.8: Example of Hydraulic erosion. This is the fractal-generated terrain in Figure 2.7 after a hydraulic erosion algorithm has been applied. Generated and rendered in GeoGen (2013)

The above methods describe a simple diffusion model, which does not accurately describe water movement and sediment transport. These are closely related to the velocity of the water. Chiba et al. (1998) introduce an enhanced method that incorporates a water velocity field. Water is placed at each vertex and the velocity of the water is determined by the local gradient. While the water flows, it dissolves some of the surface and deposits the stored sediment according to the velocity field. This improved realism comes at the cost of higher computational cost. Neidhold et al. (2005) develop a physically correct simulation based on fluid dynamics, which runs interactively, and allows for real-time manipulation of the parameters. These physics-based methods are good for improving the realism of a fractal-generated terrain but suffer from a high computational overhead.

Physics-based methods run extremely slowly when the number of points on the height-map or the number of simulation iterations increases. One way to improve the simulation time is to sacrifice physical correctness (Olsen, 2004). Another solution is to utilise modern GPUs to accelerate the simulation (Anh et al., 2007, Mei et al., 2007, Šťava et al., 2008). Despite sacrificing correctness, these methods are still difficult to control as the user can only modify a few base parameters before the simulation runs. The user also requires a fair understanding of the underlying physical laws to implement correctly. A better way to achieve realistic-looking terrain is to utilise surveyed data of natural landforms, for example DEMs (USGS, 2013).

2.2.3 Texture-based generation

Texture-based methods borrow techniques from the field of texture synthesis. Texture synthesis is a widely used technique in computer graphics for procedurally generating textures. It is used to construct a larger, possibly tileable, image from a small sample image. Textures can be arranged along a spectrum based on their properties from structured to stochastic. Structured textures are best described as possessing a repetitive, regular pattern while stochastic textures contain little structure, being close to random noise. These extremes are connected by a smooth transition as described by Liu et al. (2004). A full description of texture synthesis and its implementations is beyond the scope of research and it is only briefly discussed here as an introduction to texture-based terrain generation. We refer readers to an extensive survey on texture synthesis by Wei et al. (2009). There are two main approaches to texture synthesis: pixel-based and patch-based. Pixel-based methods generate the texture pixel-by-pixel with the new pixels value determined by its local neighbourhood. A drawback to pixel-based methods is that they tend to lose global structure. This shortcoming is addressed by patch-based methods. Patch-based methods copy and stitch blocks of pixels from the source into the output. This preserves global structure and patterns and is thus better suited to realistic texture-based terrain generation.

Balancing both user control and realism is difficult to achieve with both fractal-based and physics-based methods, due to their use unintuitive synthesis control parameters. Texture-based methods can use real height-maps (DEMs) as the source files and can thus produce highly realistic terrains. This is a recent approach to terrain generation and research on the subject is limited (Chiang et al., 2005, Dachsbacher, 2006, Saunders, 2006, Brosz et al., 2007, Zhou et al., 2007).

Chiang et al. (2005) present a patch-based system to iteratively generating macroscopic terrain based off the construction of geometric primitives by the user. A database of patches (terrain units) is manually populated by segmenting out features from real landscape maps based on two properties. The height variation for each scanline in the unit must have a higher elevation near the

centre and be lower on the boundaries. There can also only be one type of feature present, for example a hill, mountain, plain and plateau. The matching process compares the profile of the users primitive to that of the terrain units, based on cross-section, mountain ridge and terrain contour similarity. The best matching terrain unit is then orientated and translated to closely match its corresponding primitive. The selected unit is placed, such that it partially overlaps with adjacent units. The overlapped areas are stitched using a cutting method which selects pixels that minimise the elevation difference between the two units. The results from this system show boundary artefacts due to only considering horizontal height differences. Also the manual generation of the database increases the work required by the user and is limited to ridge-based features.

Dachsbacher (2006) adapts a pixel-based texture synthesis technique based on non-parametric sampling by Efros and Leung (1999). This grows a texture, a pixel at a time, by analysing the neighbourhood, which is a square window around the source pixel. Evaluating only the height value of the pixels produces unsatisfactory results because abrupt changes become visually noticeable when the terrain is rendered and artificially lit. In order to compensate for this, Dachsbacher (2006) takes into account the horizontal and vertical derivatives and this produces better results. His system allows the users to place pieces of height-maps on the work surface and have missing data synthesised. Being reliant on the technique of Efros and Leung (1999), the system suffers from long computational times but does produce compelling results. Dachsbacher (2006) suggests the use of better performing texture synthesis techniques, as well as an exploration of a patch-based approach for further the research. In general per-pixel methods do not adequately preserve underlying feature structure from the source, and this has stimulated research into patch-based methods.

Saunders (2006) present a design-by-example technique for terrain synthesis. His system utilises real-world terrain height-maps in the form of Digital Elevation Models (DEMs). Users are asked to first classify the various terrains according to their characteristics, into a logical library. This serves as the palette for the synthesis engine. The user uses this palette to describe by-example the characteristics they desire in their terrain. He achieves this during the design phase using a 2D CAD-style set of tools. Arbitrarily shaped polygonal regions are drawn in the interface and assigned a specific palette. Now the terrain is synthesised using a genetic algorithm. This algorithm is launched multiple times to generate successively higher resolution height-maps (successively finer levels-of-detail). At each level, the genetic algorithm finds a plausible way of arranging small patches of data from the respective pallets for each region. To enhance realism a border refinement operation is conducted, which is itself a subdivision operation controlled genetic algorithm. This replaces straight boundaries with short segments forming an irregular and hence less artificial-looking boundary. The output is deemed realistic as each synthesised region is statistically similar to the input files assigned to the palette. This system is said to produce an unlimited diversity of reasonably realistic terrains due to the use of a genetic algorithm. However, the actual results are no more compelling than fractal terrains after physical erosion. Further research is required to improve the visual quality of the system.

Brosz et al. (2007) present a terrain synthesis by-example system which makes use of two different terrains to synthesise a new one. The first terrain is termed the *base* and contains a rough estimate of large-scale features, such as mountains. The second is the *target* and contains high-frequency, small-scale features. The goal of the system is to extract patches of small-scale features from the target and apply them to the large-scale features of the base terrain. They incorporate an

automatic method for mapping during patch merging called Image Quilting (Efros and Freeman, 2001). It is a texture synthesis technique and starts by breaking up the base terrain into overlapping square blocks. Then for each block *i* in the base, a similar block *j* is found in the target based on feature similarity. The extended characteristics of the matched block *j* are copied back into block *i*. But instead of directly copying the data as it would be done in Image Quilting, only the details of the two patches are copied and linearly blended together. This method of blending does not produce the boundary artefacts that are common in most patch-based texture synthesis systems. The terrain synthesis system by Brosz et al. (2007) works well for combining high-frequency details into a base terrain to produce a higher resolution terrain. However, it does not create a notably different terrain from the input. As such the overall realism is closely tied to the given base terrain.

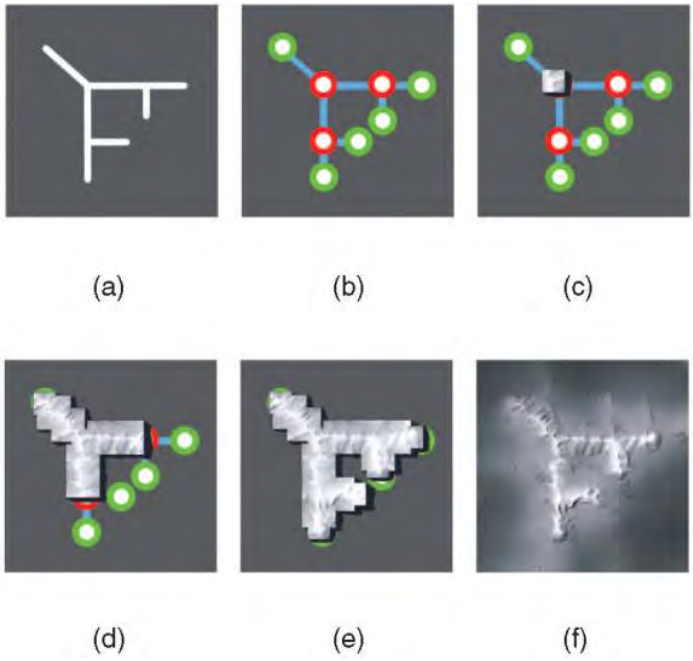


Figure 2.9: Illustration of patch placement order. (a) User Sketch. (b) Tree structure from PPA. (c) The root patch is placed first. (d) Breadth-first traversal guides placement of proceeding patches. (e) After feature placement is complete non-feature patches are placed. (f) Final result. (Image taken from Zhou et al. (2007))

Zhou et al. (2007) present a novel patch-based terrain synthesis system that makes use of DEM files and produces compelling results (Figure 2.10). The process starts with a user sketch and DEM exemplar file, the system produces a new terrain based on the exemplar's features. Figure 2.9 illustrates the algorithm and shows the three main stages:

- A User Sketch and DEM exemplar are provided to the system. These undergo a feature extraction process which identifies large-scale curvilinear features such as rivers, valleys and mountains. Zhou et al. (2007) adapt a technique borrowed from geomorphology named the Profile recognition and Polygon breaking Algorithm (PPA) which was developed by Chang et al. (1998) to identify such features. The PPA performs a breadth-first search of the input file and produces a tree structure of features. A PPA tree of the exemplar DEM is used to produce the candidate patches used in the next stage.

- The second stage controls the matching and merging of feature data into the output terrain. The PPA tree of the user sketch is broken up into patches used to order

placement. Each user patch is compared to the candidate patches and the best match is merged into the output. This process ends when all user features are exhausted.

- In the final stage, 'holes' in the output are filled by merging patches from the exemplar with no strong features.

The procedure for matching and merging is complex and discussed in detail in sections 4.2 and 4.3, respectively. The system is capable of producing a 1000×1000 terrain in approximately 5 minutes on an Intel Pentium 4 2.0 GHz with 2GB RAM. However, there are some limitations of their system, which are addressed by Tasse et al. (2011).

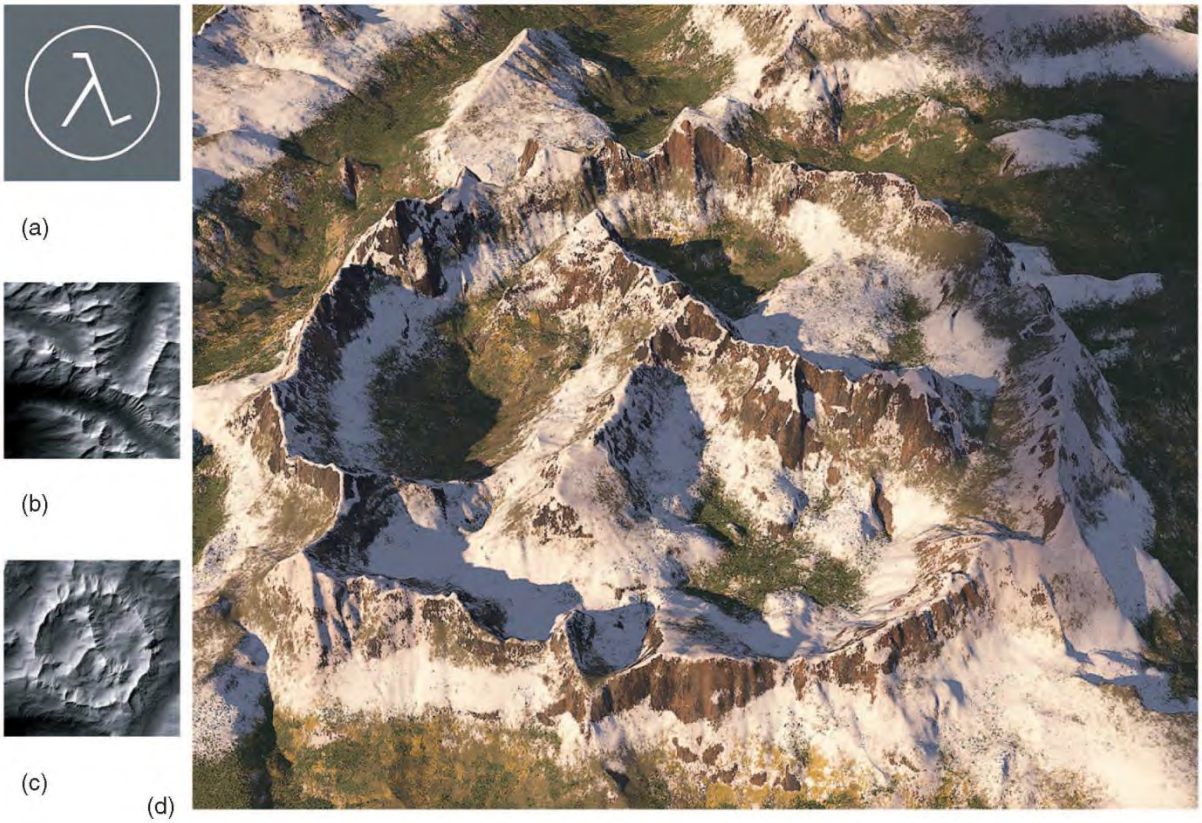


Figure 2.10: Results of synthesis. (a) User Sketch. (b) DEM Exemplar File. (c) Synthesis output. (d) Rendered terrain. (Image taken from Zhou et al. (2007))

Tasse et al. (2011) build upon work done by Zhou et al. (2007) and present improved patch merging more suitable to terrain structures that remove the visible boundary seams from overlapping multiple patches. This modified system forms the basis for the research in this thesis and we thus provide a detailed description of their system in chapter 4.

2.3 User Control

Procedural methods are designed to automate the terrain generation process with minimal user intervention. However, artists desire some level of control of the process. A trade-off must thus be made between user control and algorithm autonomy. There are several methods of user control, the most common being parameter manipulation.

2.3.1 Parameter manipulation

Parameters or variables can be used to control the generation of terrains. For example, in noise-based generation there are parameters for the amplitude and frequency of the noise function. Fractal-based methods additionally include a parameter for the roughness, which controls the irregularity of the generated surface (Fournier et al., 1982). Physics-based systems are also controlled through various parameters. These include the simulation length, strength of the erosion functions and others (Musgrave et al., 1989). The genetic algorithm used by Saunders (2006) has a large number of controlling variables and requires extensive testing to find an optimal set. The principal drawback to the use of such parameters is that the artists often do not know what the effect of changing them will have on the resulting output. Achieving a desired terrain design is likely to be an exercise of trial and error since the parameters are generally unintuitive. An improvement to parameter manipulation is the use of images to control the system.

2.3.2 Image-based control

Manually designing terrain models can be achieved by using existing 2D painting programs, for example Terragen (2013). Procedural methods can also be controlled through using images. Schneider et al. (2006) make use of images to represent the fractal base functions by providing a painting interface. Their system provides immediate feedback to the user, providing a far more intuitive form of control over arbitrary parameters. Saunders (2006) provide an authoring interface where the user describes their desired layout of terrain by painting regions using their defined palette. This is just a rough idea of where data from different terrains is to be placed during synthesis, rather than the location of specific features. Zhou et al. (2007) make use of a user-defined image which contains a simple sketch of the terrains layout to guide the synthesis. The image is made up of either black or white painted strokes which correspond to valleys and ridges, with a grey background indicating no preference. While this system can be used to specify the location of specific features, there is no mechanism for controlling the specific height. Sketching systems are primarily used by people to represent a rough design or layout. This provides a more intuitive system for controlling the generation of terrain.

2.3.3 Sketching

A sketching system is best suited for when the user has a rough idea of what the final terrain should look like. User-sketched strokes are used to specify the shape of the desired landscape. Cohen et al. (2000) present an early form of terrain sketching in their system, *Harold*. Users design and create hills and mountains by drawing 2D strokes in screen space. The endpoints of the stroke are used to create the projection plane. This is used to project the stroke into world space, creating the silhouette curve. The resulting curve forms a shadow, with points near to it being elevated based on their distance to the silhouette curve. However, the depth of the mountain (perpendicular to the screen) is constant with its cross-section being parabolic. This creates mountains that are unnatural in appearance when viewed from different angles. Watanabe and Igarashi (2004) improves on *Harold* by adjusting the depth and cross-section shape according to the shape of the stroke, resulting

in more natural looking terrain. Gain et al. (2009) present a more complex terrain sketching system based on Cohen et al. (2000). Users control the location and shape of landforms by drawing 2½D silhouette, shadow and boundary curves. These curves form constraints for a fast multi-resolution surface deformation system. During this process wavelet noise characteristics (Cook and DeRose, 2005) are analysed and applied to the resulting terrain. The synthesis system is designed to faithfully match the user's strokes rather than just approximating them, which differs from previous work. The system by Gain et al. (2009) offers a high degree of user control, allowing the user to intuitively add features such as cliffs and indentations. However, the wavelet noise fails to add small-scale natural features such as erosion patterns to the terrain. Additionally the deformations are distinctly visible when applied to an existing natural terrain. These limitations result in less realistic terrain when compared to other systems, such as Zhou et al. (2007).

More recent work by Tasse et al. (2014a), (2014b) presents a new method for editing of terrains by sketching from a first person perspective. Most sketch-based terrain systems are controlled from a top-down viewpoint, which makes it difficult to accurately describe the skyline that would be seen from the ground. The system makes use of an existing terrain, which is rendered in a 3D environment that the user can move about freely. The user will then sketch strokes to infer where terrain features should be present. These strokes are then ordered, front to back, by inferring their relative depth from the height of their end-points and detected T-junctions. Now features from the terrain, such as silhouettes and ridges, are detected. By deforming existing features the nature of the terrain is preserved as no extra features are created. The user strokes are now matched to one of these features and a specific deformation algorithm is applied and ensures that small-scale feature data is preserved. After the initial deformation the system checks that the newly modified terrain does not occlude any of the user strokes. If this issue occurs the terrain undergoes further deformation that will lower part of the terrain to remove the occlusion. This system allows a user to easily personalise an existing terrain and also preserves the style and realism.

2.4 Discussion

Based on our evaluation in this chapter, we conclude that fractal terrains lack realism while physical simulations are complex and expensive to run without extensive GPU enhancement. Both approaches also provide limited user-control. Table 2.1 provides a comparison of the three main categories of terrain generation. The "speed" entry is based on the time taken to synthesise a terrain with a size of 1024×1024 in pixels, this is a rough estimate of the speed as the figures would directly relate to the hardware being used. User-control is an expression of how easy the process is to control and realism compares the characteristics of the output terrain with real landforms. A summary of the main limitations for each category is also provided. This table shows that texture-based methods provide a high degree of realism coupled with a fair degree of user-control. We believe that realism and user-control are more important than speed of synthesis, particularly since these algorithms have not been fully optimised and there is thus room for further improvements. The content creator is more likely to wait for a longer synthesis to complete if the end result is closer to their design requirements.

	Speed	User-Control	Realism	Main Limitations
Fractal-based	Very fast	Low – High*	Low	• Absence of natural erosion • Non-intuitive control parameters • Pseudo-random output terrain
Physics-based	**Thermal:** Fast **Hydraulic:** Slow	Low	**Thermal:** Medium **Hydraulic:** High	• Complex to implement • Requires a base terrain • Minimal user control
Texture-based	Slow	Medium	High	• Limited user control • Output dependant on number of input terrains (exemplars)

Table 2.1: Comparison of terrain generation methods. *A high user-control system is provided by Gain et al. (2009)

Fractal-based techniques can run very quickly on modern CPUs but their output is unsuitable for applications where highly detailed or realistic-looking terrain is required. The synthesis is controlled with a set of parameters that do not clearly indicate their direct effect on the output, leading to a very low level of user control. Physics simulations can be used to enhance the realism of a base terrain by adding natural weathering effects. However, this also suffers from the same minimal user control and as a consequence often relies on the quality of the base terrain. The complex simulations impact on the performance of the system significantly, although recent work has focused on accelerated algorithms using GPUs. Texture-based methods borrow techniques from texture synthesis and make use of real landscapes as the source of their data, this makes the generated terrains highly realistic. When the synthesis is controlled through a sketching or painting interface, the level of user control is quite high and intuitive. The runtime is acceptable given the preference for realism but recent advances in GPU acceleration have made these methods even more appealing.

Natali et al. (2012) present a state-of-the-art report which evaluates a number of different implementations for terrain generation, which we refer interested readers to. Based on our evaluation of these terrain generation schemes we decided to extend the work done by Tasse et al. (2011). This decision is based on their improvements to work done by Zhou et al. (2007), particularly with the improved quality of merged patches. We present a detailed analysis and description of this system in Chapter 4 along with our proposed extensions to their work. The next chapter contains background information necessary to understand use of a Graphics Processing Unit (GPU) to reduce synthesis time.

3 Background: GPUs & NVIDIA CUDA

Modern *Graphics Processing Units* (GPUs) are made up of a large number of simple *Single Instruction, Multiple Data* (SIMD) processors that can be harnessed for general purpose computing. Programming GPUs has been made easier with the development of application programming interfaces (APIs) such as NVIDIA's Compute Unified Device Architecture (CUDA). GPUs have evolved into highly parallel, multithreaded, many-core processors with tremendous computational power and high memory bandwidth – Figure 3.1. NVIDIA's flagship GPU, the GeForce GTX680 attains a peak theoretical performance of 3090 billion floating-point operations per second (GFLOP/s), which is approximately 10X faster than Intel's flagship Sandy Bridge 3770K processor which peaks at 294 GFLOP/s. AMD's flagship GPU, the Radeon HD 7970, peaks at 4300 GFLOP/s representing a strong contender in terms of performance.

This chapter introduces the concepts required to understand programming on GPU devices and focuses of NVIDIA's CUDA (NVIDIA, 2013a, NVIDIA, 2013b). In section 3.2.1 we motivate our choice for CUDA over other alternatives. The remainder of the chapter provides information on the programming model and execution pipeline for NVIDIA GPUs.

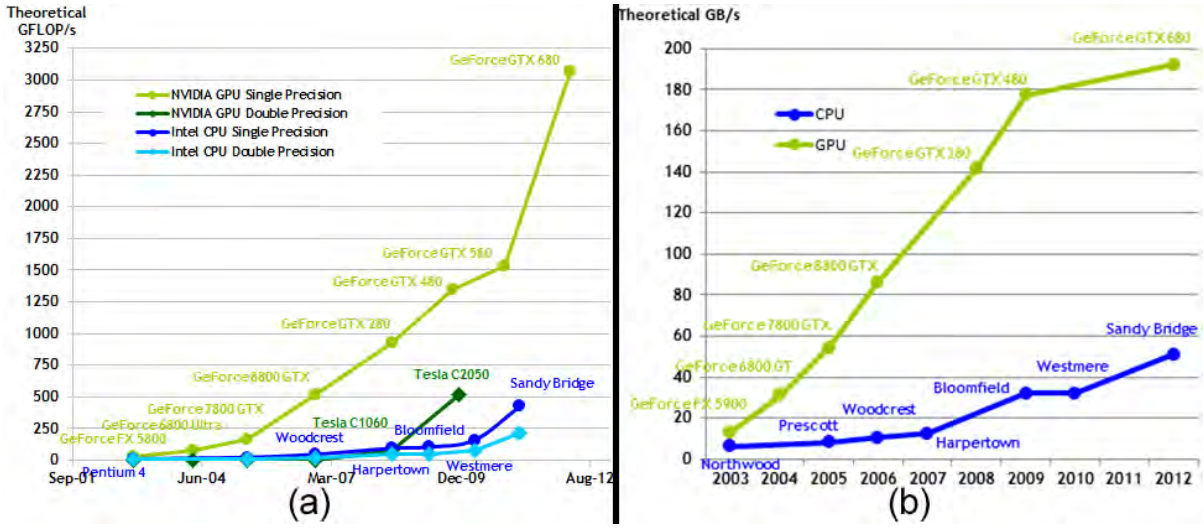


Figure 3.1: (a) Floating-Point Operations per Second and (b) Memory bandwidth, for both CPU and GPU (NVIDIA, 2013b). This shows the large difference between GPU and CPU performance leading to the use of GPUs for accelerated computation.

3.1 GPUs and Parallel Programming

Graphics accelerators were the precursor to what we now call GPUs and have been in use in computer systems since the early 1980's principally to accelerate drawing operations. For the last two decades the development of graphics hardware has been extensively driven by the gaming industry since games required more vector-parallel processing power than typical CPUs could offer. CPUs feature a complex processing architecture that cannot keep up with the large number of fragment operations required to render complex 3D graphics efficiently. This led to the development of specialised hardware that contains large numbers of simple processing units designed to efficiently process large amounts of fragment data. Early GPUs (1990s) featured a fixed-function rendering pipeline where specialised hardware components were dedicated to individual stages. The

user controlled the rendering process by configuring parameters such as vertex positions and colours of vertices or lights. Because rendering functions were predicated on the availability of compatible hardware, the use of a fixed-function pipeline evolved into a programmable one in 2001. This enabled the control of vertex and fragment operations through small programs called *shaders*. GPUs follow a different development philosophy from CPUs and focus more on parallel computation over reduced memory latency. For example a CPU would process vertices and fragments sequentially whereas a GPU can process multiple elements at the same time. Although modern CPUs contain multiple cores leading to a small degree of parallelism, however, this is dwarfed by the large number of cores GPU devices contain. GPUs adopt a SIMD approach, which utilises a large number of simple processors to execute the shader programs in parallel. Shader languages such as the DirectX High-Level-Shader-Language (HLSL), NVIDIA CG and the OpenGL Shading Language (GLSL) can be used to write shaders for the GPU and facilitated the development of more advanced rendering techniques than allowed by a fixed-function pipeline, such as bump-mapping. These languages were initially used to write shaders for general purpose GPU computing but they are inherently designed for graphics operations, such as filtering and rendering. The use of specific graphics terminology made programming on GPUs inaccessible to typical programmers interested in more general acceleration and this led to the development of a high level APIs such as CUDA. CUDA was released in early 2007. It gives developers access to the virtual instruction set and memory of these devices, allowing for general purpose GPU (GPGPU) programming. This followed NVIDIA's launch of the GeForce 8 series in 2006, which featured a generic *stream processor* enabling the GPU to act as a more generalised processing device.

Figure 3.2: GPU devotes more transistors to data processing (NVIDIA, 2013b). There are significantly more Arithmetic Logic Units (ALUs) dedicated to the control and cache units.

Modern GPUs dedicate more transistors to data processing rather than data caching and flow control (shown in Figure 3.2). This explains the large discrepancy in floating-point capability between the CPU and GPU (Figure 3.1). Because there are fewer transistors dedicated to caching, memory latency typically becomes the main bottleneck of these systems. Consequently, GPUs are well-suited to address problems that can be expressed as data-parallel computations. This means the same program is executed on many data elements in parallel with a high *arithmetic intensity*. Arithmetic intensity refers to the ratio of computation operations to memory operations. Unlike CPUs, GPUs have a parallel throughput architecture, which emphasises executing many concurrent threads slowly rather than a single thread quickly. Applications that require processing of large data sets can use a data-parallel model to speed up computation, specifically those that have a high arithmetic

intensity. Apart from image processing and 3D rendering, other algorithms ranging from physics simulations to computational finance can be expressed as data-parallel computations and greatly accelerated. This has led to a great deal of research into exploiting GPU devices. The programming of these GPGPU devices is achieved through the use of APIs such as NVIDIA's CUDA but other alternatives exist such as OpenCL (Khronos, 2013). These are discussed in section 3.2.1.

3.2 NVIDIA CUDA

NVIDIA released the initial version of its CUDA SDK in early 2007. This has evolved considerably over the years but remains limited in availability, as only NVIDIA GPUs are supported. CUDA programs are mostly written in C/C++ which gets compiled by NVIDIA's *nvcc* compiler. Other languages such as Python and Java are supported by 3rd party wrappers. The CUDA API is better suited to general programming than shader languages and has several advantages including scattered memory access, exposed shared memory, and full support for integer and bitwise operations. CUDA has continued to evolve as the underlying hardware architecture evolves and new features are implemented. The use of these new features is linked with the *Compute Capability* (CC) of the device, with higher level devices being fully backward compatible. At the start of our research the GeForce 5xx series of devices were available with Compute Capability 2.0. As such further discussion in this chapter is focused around the features available for this version.

3.2.1 Motivation for using CUDA over alternatives

NVIDIA CUDA is not the only high-level API for GPGPU programming. There is the now deprecated ATI Stream (ATI, 2013) and the open-source OpenCL (Khronos, 2013). ATI released the first version of Stream at the end of 2007 but subsequently deprecated this in favour of development in OpenCL. It was developed explicitly for their Radeon series and was restricted to AMD GPUs. OpenCL, in contrast is an open-source cross-platform framework that allows for execution of parallel programs across heterogeneous hardware platforms. The support and performance of OpenCL has improved steadily since its first release in 2008. However, despite all the improvements, CUDA paired with compatible NVIDIA GPU still has a performance advantage over an OpenCL implementation (Karimi et al., 2010, Du et al., 2012) and also has better support for 3rd party libraries. Intel recently released Xeon Phi (Intel, 2013), a hardware coprocessor comprised of many-core processors, set to rival the use of GPUs for parallel computing. It went into first production in late 2012 and the initial variants have a significantly higher price point than consumer grade GPUs and as yet not much support is available for development. We expect this to change over the coming years but was not available for consideration during our research.

This left a choice between an OpenCL and CUDA implementation for GPU acceleration in this work. During the planning stage for our research, CUDA provided better performance and had greater support than OpenCL. This led to our decision to develop a CUDA based solution for GPU acceleration.

3.2.2 CUDA Programming Model

CUDA supports a C-like syntax which eases the transition for existing programmers without requiring a graphics background. This language is also easily interoperable with standard C and C++ code. CUDA code gets compiled into a special format that is deployed at runtime to the CUDA-capable device where the code is executed by thousands of lightweight threads. These threads are divided up amongst the devices many compute cores. CUDA claims to run on any CUDA capable

device, which is not entirely true. New hardware features are introduced in the form of the devices Compute Capability (CC). This is fully backwards compatible, which means that as long as the device has the minimum required CC, the code will execute.

There are two hardware abstractions that CUDA defines: the *device*, which is a CUDA capable GPU, and the *host*, which is the computer to which the device is connected. The execution of code on the device is initiated by invoking a C-like function called a *kernel*. Before the kernel is called, data needs to be transferred to the device's memory. The kernel then executes simultaneously on the many threads of the CUDA device, processing the data. After execution is complete the data can be transferred back to the host. This is illustrated in Figure 3.3.

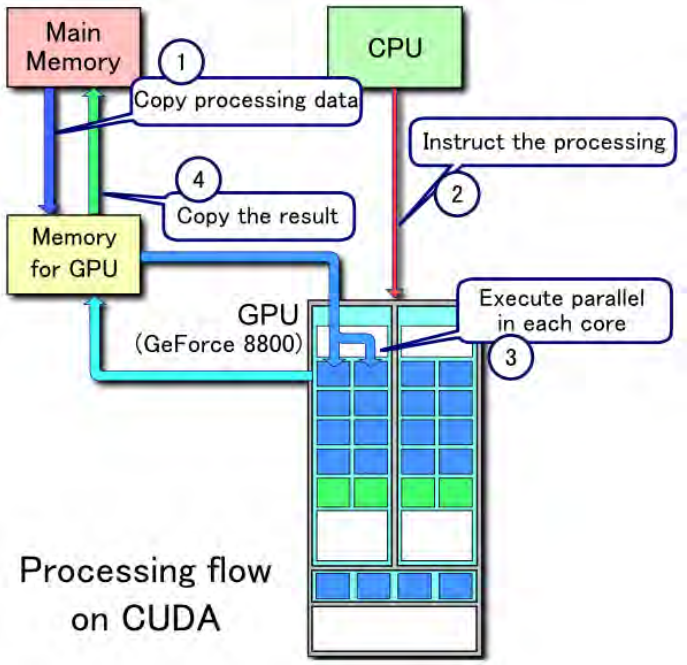


Figure 3.3: CUDA Processing Flow. (1) Data is copied from host to device; (2) Kernel is executed; (3) Data is processed in the many threads on the GPU; (4) Result is copied back to host.

There are two important high-level aspects to CUDA programming: memory management (how data is transferred between host and device) and the execution pipeline (how data is processed on the device). CPUs contain multiple levels of cache which hides memory access latency and ensures the processor is always being utilised. This is not the case for GPUs, which rely on many lightweight threads and instantaneous context switching to swap out threads waiting for memory and swap in threads ready for processing. We now provide further explanation of these aspects.

3.2.3 Execution Pipeline

The key to understanding the CUDA execution pipeline is to understand how the GPU hardware and software components interact and how instructions are scheduled. NVIDIA was careful to design a software model so that it mirrors the characteristics of the underlying hardware. This means the programmer can design their system such that it maps as closely as possible to underlying hardware. Multiple schedulers provide fast switching of the many executable threads. The threads are grouped into blocks and *blocks* are arranged into a *grid*. The GPU contains a number of *Streaming Multiprocessors* (SMs) that are assigned a number of blocks from the grid. The SM is responsible for scheduling the execution of the threads for the blocks it is allocated. However, the user is able to

fully control the thread and block layout. Appropriate design of these layouts can allow the programmer to utilise the GPU more efficiently.

Software – The Grid, Blocks & Threads:

As of compute capability 2.0, the CUDA code is executed on a GPU device by launching a kernel, which is a C-like function call. Parameters to this include the dimensions of the grid and block and number of threads per block. The code within a kernel is divided up into smaller logical units called blocks and has up to three-dimensions with maximum number of $(65535 \times 65535 \times 65535)$ blocks and are collectively known as the grid. In turn each of the blocks comprises of up to three-dimensions, with a maximum of $(1024 \times 1024 \times 64)$ and up to 1024 resident threads per block. Resident threads refer to how many active threads can be instantiated within the block based on the available resources for storing the threads context data. An example of this layout is shown in Figure 3.4. Each block on the grid is executed independently of the others with no guarantee on the order of block execution and no mechanism for inter-block communication. The GPU scheduler controls which block is being executed based on its availability of warps (discussed later – Block, Warp & Thread Scheduling) ready for processing and enforcing cooperation could result in blocks being stalled or a case of system deadlock. However, threads within the same block can communicate through the use of shared memory (see section 3.2.4) and synchronisation (discussed later – Barrier Synchronisation). There are a maximum of eight resident blocks per SM, and each SM is also limited to a maximum of 1536 schedulable threads. The programmer needs to adhere to these constraints in order to maximise device throughput. By adapting the computational problem to utilise all of the available threads, the programmer can ensure the GPU is fully saturated with work.

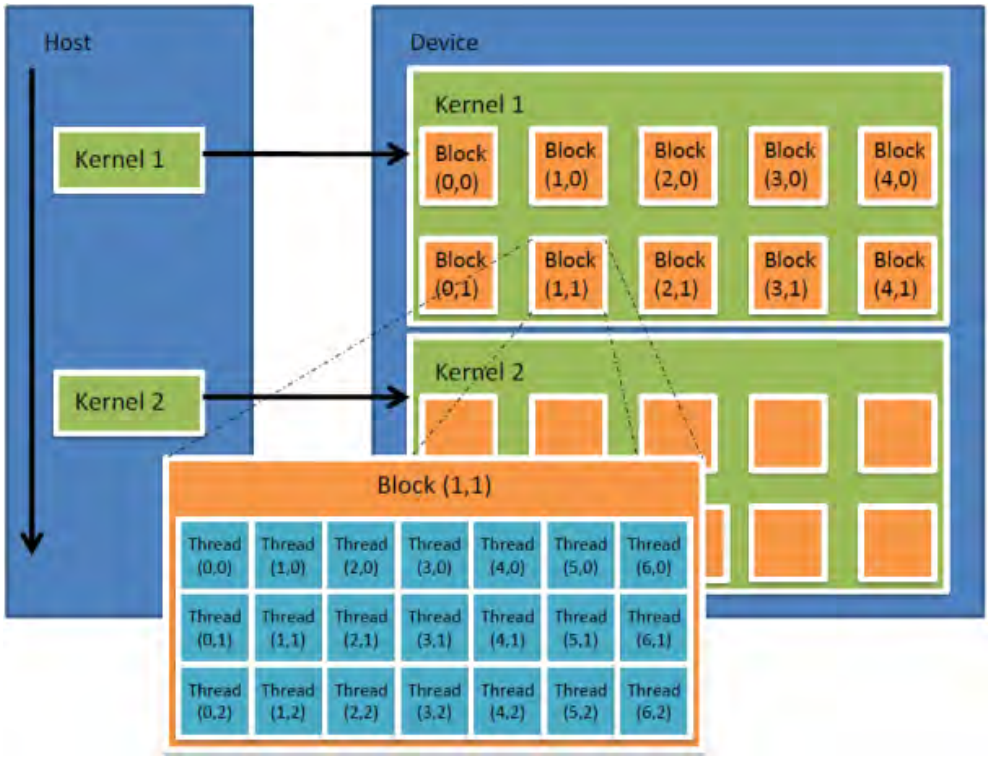


Figure 3.4: Schematic overview of the Grid-Block-Thread layout (NVIDIA, 2013b). The kernel is loaded onto the device which is comprised of the blocks and threads.

Layout & Indexing of Blocks & Threads

Blocks can be arranged in one, two and three-dimensions on the grid. A block supports up to three-dimensional thread layouts. An example is presented in Figure 3.5 which features a grid with dimensions of (3×2), which holds blocks of dimension (4×3). This gives a total of 6 blocks within the grid, each with 12 threads. Grids support a maximum of 65535^3 blocks, but since a single block is executed in a single SM, these serve more as a programming convenience for organising the problem. The dimensionality of threads within a block is more important in maximising throughput of the device. However, the layout of threads within a block can have a significant impact on performance.

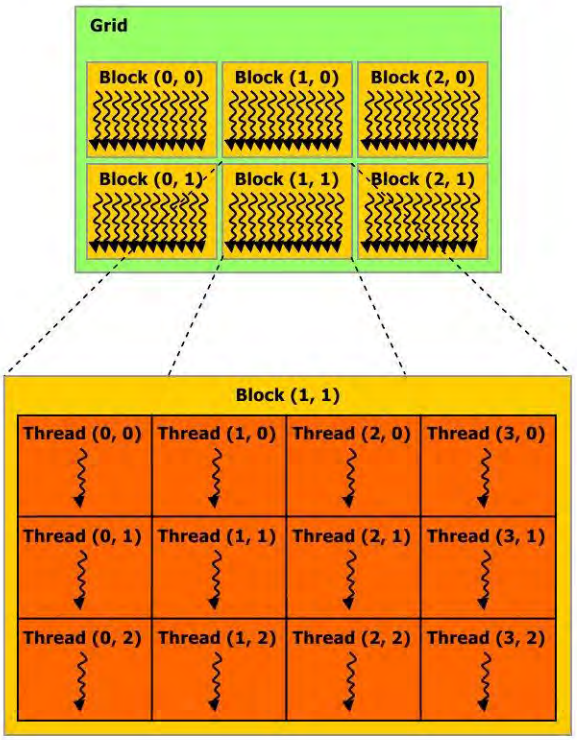


Figure 3.5: Example Grid/Block/Thread Indexing for a 2D grid and block layout (NVIDIA, 2013b).

Thread indexing is a vital part in controlling the specific execution path and specifying what data the thread is to operate on. When a kernel is executed, the defined number of threads will all execute the same code. CUDA provides kernel variables that can be used to determine the index of the thread and block, namely *threadIdx* and *blockIdx* respectively. Along with these are blockDim and gridDim, which store the defined dimensions of a block (threads in a block) and grid (blocks in the grid). Listing 3.1 provides a simple example of a CUDA kernel which squares the values of a (512×512) array. The kernel invocation is provided in Listing 3.2 where the dimensions of the blocks and grid are defined. In this example, blocks have a dimension of (32×32) which produces a total of 1024 threads. In order to evaluate all values in the array, a grid size of (16×16) blocks is required. CUDA operates on one-dimensional arrays, meaning multi-dimensional arrays need to be *flattened*. After completion, all the values in the array will be squared.

```
__global__ ExampleKernel(float* data, int data_w)
{
    // Use index of thread to work out index in array to access
    int idx_X = threadIdx.x + (blockIdx.x * blockDim.x);
    int idx_Y = threadIdx.y + (blockIdx.y * blockDim.y);

    // Calculate the flattened index (1D)
    int idx_Flat = idx_X + (idx_Y * data_w);

    // Read the value from the data array
    float val = data[idx_Flat];

    // Write back the value squared
    data[idx_Flat] = val * val;
}
```

Listing 3.1: Example of a CUDA Kernel. This kernel takes a flattened square array of size w and squares its values.

```
int main(void)
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(32, 32);
    dim3 blocksPerGrid(16, 16);
    ExampleKernel<<blocksPerGrid, threadsPerBlock>>>(data, data_w);
    ...
}
```

Listing 3.2: Example Kernel Invocation. This is the sample code which will launch the CUDA kernel defined in Listing 3.1. The threads-per-block and blocks-per-grid are defined and used in the call. This also assumes initialisation of data for the array on the device.

Hardware – The GPU, Streaming Multiprocessors & Cores:

The CUDA software model arose directly from the design of the CUDA hardware. A CUDA capable device is made up of a collection of Streaming Multiprocessors (SMs), which contain the CUDA cores, memory infrastructure and control units as shown in Figure 3.6. There is a direct mapping between the software and hardware models: grid to GPU device, block to a SM and a thread to a single core.

Each SM includes 32 *Scalar Processors* (SPs) or CUDA cores, 16 *Load/Store Units*, 4 *Special Function Units* (SFUs), Dual *Warp Schedulers* and *Dispatch Units*, 32k 32-Bit registers and 64KB of combined Shared Memory and L1 Cache (NVIDIA, 2013c). The SM is assigned a group of blocks (by the GPUs scheduler) which it operates on in turn, with the threads defined by the block being executed on the cores. The 16 load/store units allow for source and destination addresses for 16 threads to be calculated per clock-cycle, thus requiring only two cycles to access memory for all the cores. The 4 SFUs execute transcendental instructions such as sin, cosine, and square root. Execution in the SM is performed in a group called a *warp* (discussed later – Block, Warp & Thread Scheduling), with each warp being comprised of 32 threads. Each SFU executes only one instruction per thread, per clock, requiring 8 clock-cycles to complete a warp. These SFUs are decoupled from the dispatch unit allowing it to issue instructions to other execution units while the SFUs are occupied. Dual Warp Schedulers select an instruction from each warp and issue them to a group of 16 cores, 16 Load/Store units, or 4 SFUs which are executed independently. The fast on-chip memory provides limited L1 cache and method for threads to cooperate. The architecture allows this memory space to

be divided into 16KB of L1 cache and 48KB shared memory, or 48KB L1 cache and 16KB shared memory depending on the requirements of the programmer.

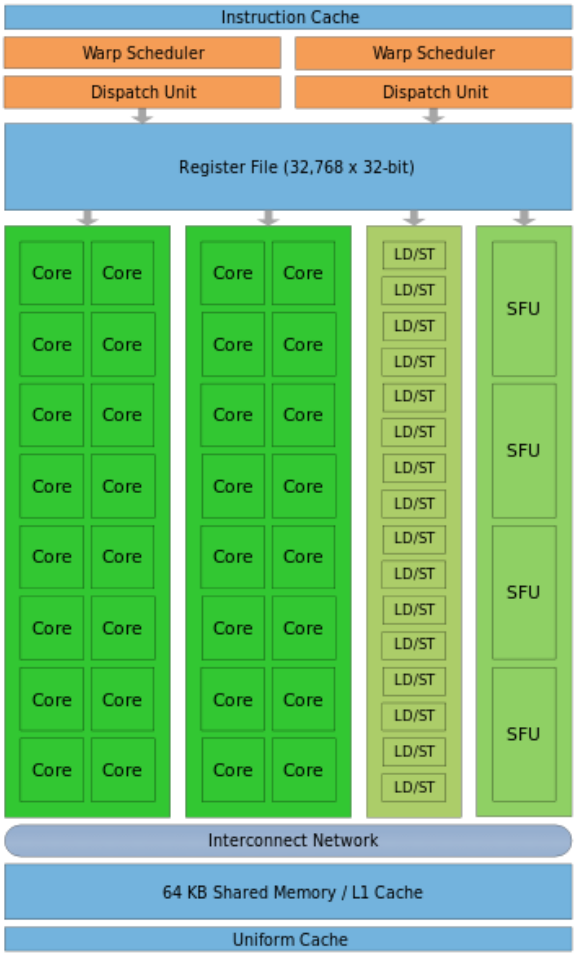


Figure 3.6: Architecture of a Scalar Multiprocessor unit for a GeForce GTX 580 (Fermi) GPU (NVIDIA, 2013c). This represents all the command, control and cache units present.

CUDA is designed to operate with any number of SMs, as the total number of blocks is divided amongst all available SMs. This produces a highly scalable model since adding more SMs to a device increases computational throughput seamlessly. This is usually the principal difference between low-end and high-end devices. High-end GPU devices usually carry a larger number of SMs and larger amount of device memory.

Block, Warp & Thread Scheduling

When developing the kernel code, the programmer statically defines the number of blocks, threads-per-block along and the dimensions of both. These are used when the kernel is invoked for the hardware's block scheduler to divide up and assign blocks to the devices SMs. Each SM breaks up its assigned blocks into groups of 32 threads called warps. Each SM can have a maximum of 8 resident blocks and 48 resident warps, with others waiting in a queue. The number of warps per block is calculated by taking the number of assigned threads (t) and dividing by the warp size (32).

Fermi devices have two warp schedulers which allows for each to select an instruction to be run concurrently and independently. The hardware is mapped into two sets of 16 cores, 16 load/store

units and 4 SFUs. Each of the warp schedulers can utilise one of these items at any given time. There is also no requirement that warps need to be from the same resident block. However, each of the instructions belonging to a particular warp need to be executed in order. Each resident warp (w) has its own context (instruction counter and registers). This means that instruction 7 of warp w_1 can execute, then on the next clock-cycle instruction 3 of warp w_5 can execute, followed by instruction 8 of warp w_1 immediate after. This *instantaneous context switching* differs from that of a CPU which has a heavy-weight thread context, and incurs a delay due having to copy register data in and out of the CPU. CUDA allows for this instantaneous switch, as only a switch to the next scheduled warp's instruction pointer and registers is required, both of which are store in on-chip memory.

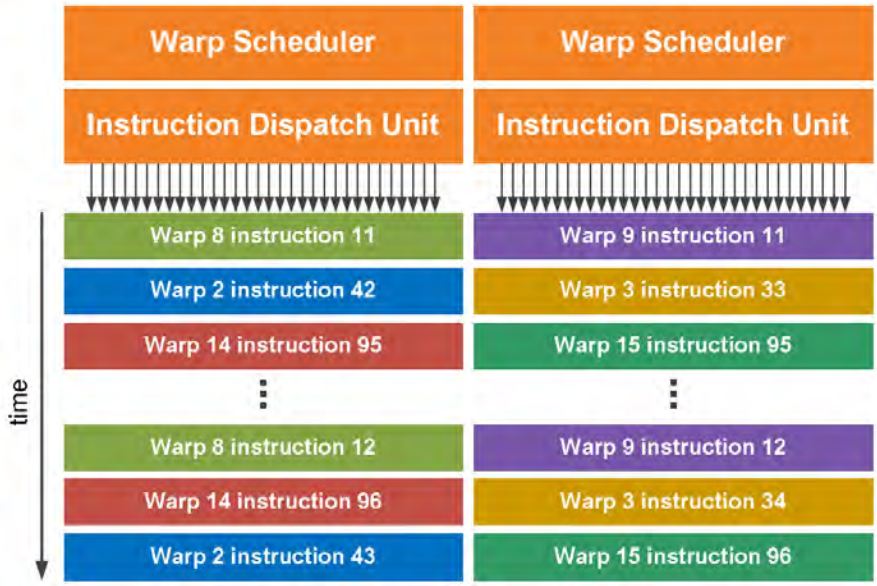


Figure 3.7: Example of Fermi's Dual Warp Schedulers. Each scheduler is assigned a group of warps; the first scheduler is responsible for warps with positive ID and the second for negative IDs. At each clock-cycle both the schedulers select an instruction to execute for a particular warp. Since two warps are run concurrently, each works on only half its instructions, requiring two cycles to complete. (NVIDIA, 2013c)

Having the ability to instantaneously switch between warps is used in hiding memory latency, so that when a warp makes a memory request it can be switched out for one awaiting execution. This is however, predicated on the fact that there is another warp available for execution. With this in mind, the programmer should design the system to provide enough warps in order to saturate the device.

An example of how the dual warp schedulers work is presented in Figure 3.7. During the first clock-cycle, instruction 11 from warp 8 and instruction 11 from warp 9 are executed. At the next clock-cycle, instruction 42 from warp 2 and instruction 33 from warp 3 are executed. This shows that two different instructions from two different warps are executed concurrently in the hardware. But since there are 32 threads in a warp and only 16 execution units, both sets of instructions execute twice over two clock-cycles, one for each batch of 16 threads.

Flow Control & Code Divergence

Flow control refers to the use of a control instruction (if, switch, do, for, while) in the execution of a kernel. Using these controls can significantly impact the instruction throughput by causing threads within a warp to diverge (follow different execution paths). In this case, all the required execution

paths are serialised and evaluated by all threads in the warp, increasing the number of instructions executed. This means for an 'if' statement, the 'true' branch is evaluated first then the 'false' branch. They are not executed concurrently as is the case with traditional multi-core CPU systems. Threads that diverge down the 'true' branch will ignore execution when executing the 'false' branch, and the 'false' threads ignore execution in the 'true' branch. After all the paths have been followed, the threads converge back to the same execution path.

However, if all the threads in the same warp follow the same execution path (i.e. all threads evaluate to 'true' or 'false'), then only the required branch is executed. If even a single thread diverges within a warp then both branches will be executed. This is a result of CUDA using a lock-step mode of execution within a warp, where all threads can only perform the same instruction at any given clock-cycle. The programmer needs to take this into account when using flow control statements so as to avoid the possibility of divergence.

Barrier Synchronisation

Synchronisation within a kernel allows the kernel to halt until all threads in the same block reach the barrier. This technique, together with shared memory, allows for cooperation of threads within a block. Synchronisation is limited to block level and, unlike multithreading on a CPU, it is impossible to synchronise all threads across the device. Synchronisation is useful when using threads to load data from global memory into shared memory before computation so as to make sure data is available. A downside to using synchronisation is that threads blocked at a barrier are idle, which reduces the overall performance of the device.

3.2.4 Memory Hierarchy

Since GPUs focus more on parallel data processing rather than memory caching, maximising the memory throughput is essential to maximise performance. For example, a GeForce GTX 580 has a peak memory bandwidth of 192.4GB/s but the host-to-device transfer over the PCIe x16 Gen2 bus peaks at a comparatively low 8GB/s. This means that transfers to or from the device should be limited and calculations should be executed on the device, even if they would be faster on a CPU as the memory transfer performance penalty would dominate processing time. For example, before squaring the values of a 10,000 element 2D array of floats, 0.046s is required to transfer data to host, calculated as follows $\left(1s \div \left[8GB/s \div \left[10000^2 \times 4B\right]\right]\right)$. On top of this the computation cost for executing on the CPU needs to be added, which further increases the total computation time. However, processing the array on the GPU only requires 0.0019s of compute time. This represents a 24x performance drop in transferring the data alone. CUDA has access to much of the memory present on the GPU. The scope and characteristics of these memories are summarised in Table 3.1.

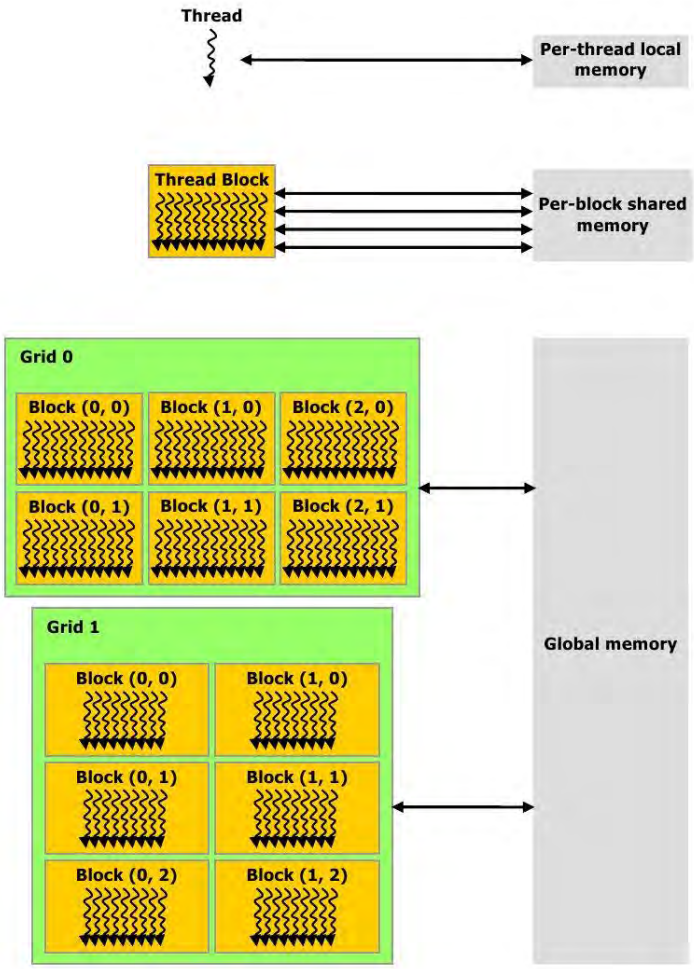


Figure 3.8: Memory Hierarchy. Each level shows the scope of the different types of memory. Local memory is restricted to a single thread. Shared memory can be accessed from all threads in a single block and global memory is accessible between one or more grids. (NVIDIA, 2013b)

There are two fundamental categories of memory: on-chip and off-chip. On-chip memory is very fast with near zero latency for access but is very limited (64KB per SM). It includes both registers and shared memory. Off-chip memory has a far higher latency, but is much larger (typically up to 3GB) and includes global, local, constant and texture memory. Every level of the execution pipeline has a corresponding memory space, as shown in Figure 3.8. Each memory type has a specific use and must be assigned based on the problem being solved in order to maximise performance of the system. The different memory types are discussed below:

There are 32k 32-Bit registers located on each multiprocessor that get shared between all its cores. Accessing registers consumes zero extra clock cycles per instruction, but delays can occur if there are read-after-write dependencies and bank conflicts. A bank conflict occurs when two or more addresses of a memory request fall within the same bank. Information on bank conflicts can be found in section 3.3.1.The read-after-write latency is approximately 24 cycles, which is how long a thread is required to wait before accessing that register again. This latency can be hidden if at least $24 \times n$ threads are active in the multiprocessor with n being the number of cores present. The hardware scheduler attempts to optimally schedule instructions to avoid register bank conflicts and

works best when using a multiple of 64 threads per block. Should there not be enough registers in the SM or if the value is too large to store, the data will spill over into local memory.

Type	Location	Cached	Access	Scope	Lifetime
Register	On-Chip	N/A	r/w	Thread	Thread
Local	Off-Chip	*Yes	r/w	Thread	Thread
Shared	On-Chip	N/A	r/w	Block	Block
Global	Off-Chip	*Yes	r/w	Global	Application
Constant	Off-Chip	Yes	r	Global	Application
Texture	Off-Chip	Yes	r	Global	Application

Table 3.1: Device Memory Summary. *Cached on devices with Compute Capability 2.0 and up.

Local memory resides in off-chip device memory and suffers from high latency and low bandwidth. Up to 512KB may be allocated per thread. The compiler automatically assigns local memory for large structures; arrays that would consume too much register space and "spilled over" registers in the case the thread runs out of its allocated amount. Local memory accesses are always stored in L1 and L2 cache and should be avoided as the latency is still very high.

Shared memory is located on-chip and can be accessed by all threads resident to the block currently loaded onto the multiprocessor. It has a much higher bandwidth and lower latency compared to global memory. Each SM has 48KB which is divided up into 32 distinct 32-Bit memory banks that can each be accessed simultaneously within a warp. This means that 32 threads can each make a request to shared memory without bank conflicts, provided multiple requests do not fall in the same bank. Shared memory can either have all the threads read from the same memory address or all read from unique addresses to avoid a bank conflict. Shared memory is useful for problems that require threads to co-operate or when a larger problem is divided up between the threads to solve a smaller component. The use of shared memory can greatly improve system performance over using slower global memory.

Global memory is the largest pool of memory that is located off-chip and has a size up to 3GB that can be read from and written to by all threads, blocks and the host system. Global memory is the device equivalent of the host's RAM. The downside to using global memory is the high latency associated with reads and writes. This high latency can be offset if a coalesced memory read is performed. Coalesced access occurs when all the threads within a warp (32 threads) can be combined into as little as one memory read. Global memory consists of rows of 128-Byte (128B) aligned segments that are accessed in 128B transactions. This allows for all threads in a warp to access adjacent 4B words (float) in a single request provided the memory is aligned to a single cache line, even if the reads are non-sequential. However, for misaligned access, two separate requests are required to read all the memory for the threads. These different patterns are represented in Figure 3.9. The correct use of coalesced memory access for reading or writing data greatly improves memory throughput on the device.

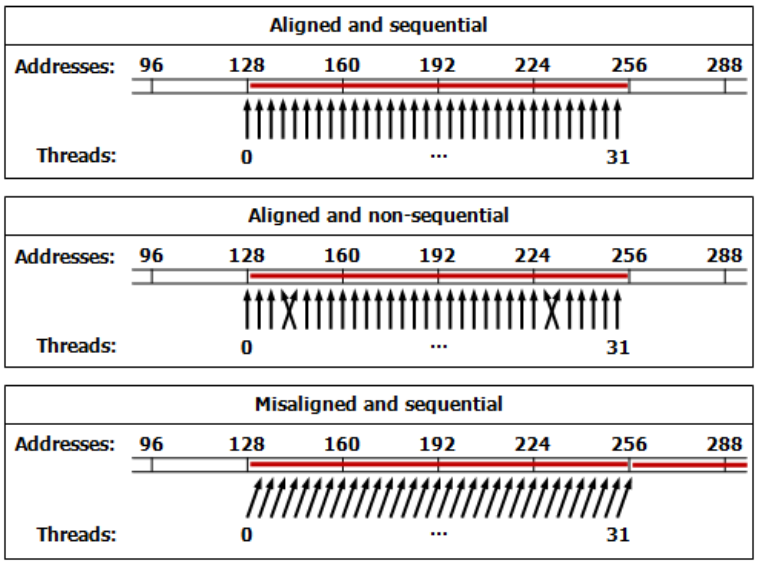


Figure 3.9: Memory access pattern for coalesced reading. Both (a) and (b) require a single 128B transaction whereas (c) requires two 128B transactions, which decreases performance to 50%. (NVIDIA, 2013b)

Constant memory is read-only off-chip memory stored in constant cache with a total size of 64KB. It is mostly used by the host to set constant values that need to be read across multiple kernel executions. Reads from constant memory cost one request from a register but only when all threads are reading the same address. If different addresses are requested by threads within a warp then the requests get serialized and the cost scales linearly with the number of different addresses read.

Texture memory is a read-only off-chip memory, which is part of global memory. The difference between this and global memory is that it is spatially cached in one, two or three dimensions. With the data being spatially cached and a lookup operation is performed on a memory address, the adjacent memory locations, both vertically and horizontally, are also fetched and stored in L1 cache on the device. This speeds up access when an operation on a given location requires the data stored in neighbouring locations, such as image filtering. However, in the event of a cache miss the cost is one read from device memory, negating the advantage of using texture memory. It is primarily used when accessing two-dimensional data-structures where adjacent memory reads are required for computation.

These memory types each have their advantages and limitations. Selecting the right memory type for the specific problem at hand is essential in obtaining higher performance from the device. Higher memory throughput means that less time is spent waiting for data.

3.3 Performance considerations

In order to exploit the high computational power of modern GPU devices three key optimisations are required (NVIDIA, 2013a):

3.3.1 Maximise memory throughput

Bank conflicts occur when two or more threads make an access request to the same memory bank. If this occurs, the request gets serialized and the hardware splits the request into as many conflict-free requests (n) as needed. This decreases memory throughput by a factor of n as a result. However, if all the threads request the same address then a broadcast is performed. This is when the

address is read only once and broadcast to all the requesting threads. Bank conflicts should be avoided to maximise memory throughput.

The amount of data being transmitted between the host and device should be minimised as the bus transfer speeds are significantly slower than transfers within the device as shown in section 3.2.4. Another consideration is to select the correct memory type for the problem. Global memory is larger but has the highest latency, although this can be reduced by using memory coalescing. Shared memory provides fast access provided there are no bank conflicts and that the data is small enough to fit.

3.3.2 Maximise parallel execution

The ratio of parallel to sequential computation must be balanced. Asynchronous calls should be used where possible to enable concurrent execution between the host and device. Asynchronous calls allow the host to transfer data to the GPU and continue processing. Once the data has been transferred the kernel can be executed and the host can continue processing and initiate more asynchronous calls. This allows for both devices to be continuously executing data in order to maximise performance.

3.3.3 Maximise instruction throughput

The use of arithmetic instructions with low throughput should be avoided as larger throughput hides memory latency. Branch divergence within a warp must be avoided as threads in a warp cannot execute different code concurrently, requiring multiple passes to execute all branches. Reducing unnecessary instructions and optimising away synchronisation points will also increase instruction throughput.

3.4 Summary

Three key optimisations are required to exploit a GPUs full potential:

- Maximise memory throughput
- Maximise parallel execution
- Maximise instruction throughput

All the required hardware details should now be understood so that the implementation chapters can be easily followed. Both chapters 6 and 7 include sections on our specific GPU implementations. It is important to understand that GPU programs are defined by the launching of kernels, which when executed are split up into many threads, blocks and grids. Knowledge of how these threads are executed on the hardware is important in order to optimise the system to maximise performance. Since different hardware generations provide different feature-sets, the compute capabilities must be understood. Our system was designed around an NVIDIA GTX 580, which has a compute capability of 2.0.

4 Framework

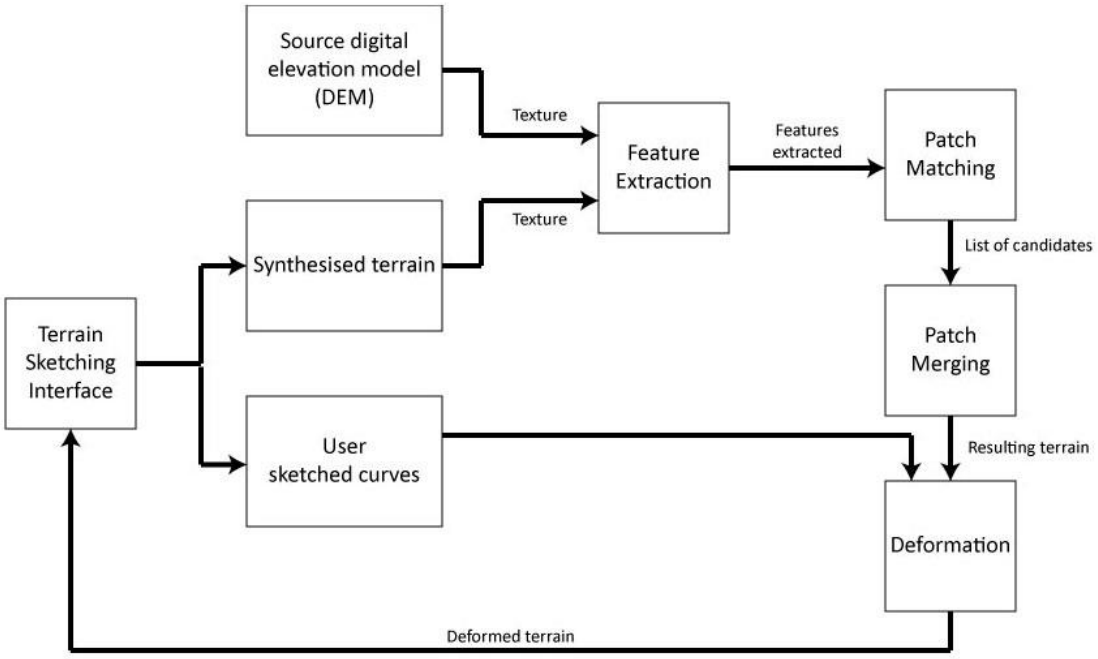


Figure 4.1: Overview of patch-based terrain synthesis framework developed by Tasse et al. (2011). The terrain sketching interface is the entry point to the system, where the user sketches their desired terrain. This is used initially to produce a synthesised terrain, which together with a source file is run through feature extraction. Patch matching and merging is run with the result being deformed according to the user's initial sketch to produce the final terrain. This feeds back allowing the user to modify the terrain and re-run synthesis.

This thesis extends the work done by Tasse et al. (2011), which is a patch-based terrain synthesis system based on Zhou et al. (2007)'s work. An overview of Tasse et al. (2011)'s system is presented in Figure 4.1. The authors improved on several aspects of Zhou et al. (2007)'s algorithm, discussed in section 2.2.3, to enhance the quality of generated terrain. The components shaded in grey represent the core terrain synthesis steps and are explained in this chapter.

4.1 User Input & Feature Extraction

Zhou et al. (2007) provides users with a limited 2D sketching interface. This only allows users to specify the 2D position and type of feature, ridge or valley, but allows no control over the height of these features. Tasse et al. (2011) develop a new hybrid scheme that makes use of the sketching interface from Gain et al. (2009). Users are presented with an interactive environment that allows them to sketch $2\frac{1}{2}$D constraint curves. These curves describe the paths and types of features as well as allowing the user to sketch out the height profile. This information is used during different stages of the synthesis engine; the height profile is used to deform the terrain after it has undergone merging and matching. Firstly the user's sketches are converted into a 2D map containing the ridge and valley curves to be used during feature extraction. An example of a user sketch is provided in Figure 4.3 (a).

Feature extraction works by automatically identifying features, such as ridges and valleys, in the target terrain. In image processing, features are typically identified through the use of edge-

detection methods. These features are characterised by the locally maximal derivatives of the image intensity, whereas terrains are based on local extrema of the height-map. Naively applying the same techniques to terrain feature extraction results in spurious features due to local height variations (Zhou et al., 2007). Thus a method making use of local extrema in the height-map is required to extract the terrains features. The Profile Recognition and Polygon Breaking Algorithm (PPA) (Chang et al., 1998, Chang and Sinha, 2007) is designed for this purpose. The original PPA inefficiently breaks cycles by removing the largest value edge and runs in polynomial time with respect to the number of edges. Bangay et al. (2010) explicitly reformulate the PPA as an equivalent process with the elevation data represented as a graph and computes the *minimum spanning tree* (MST). The MST of a graph is the subset of edges which allow the graph to remain connected and minimise the total weight along all the edges, commonly created with greedy algorithms such Kruskal (1956) and Prim (1957). The PPA comprises of the following 5 basic steps: Profile recognition, Target connection, Polygon breaking, Branch reduction and Line smoothing.

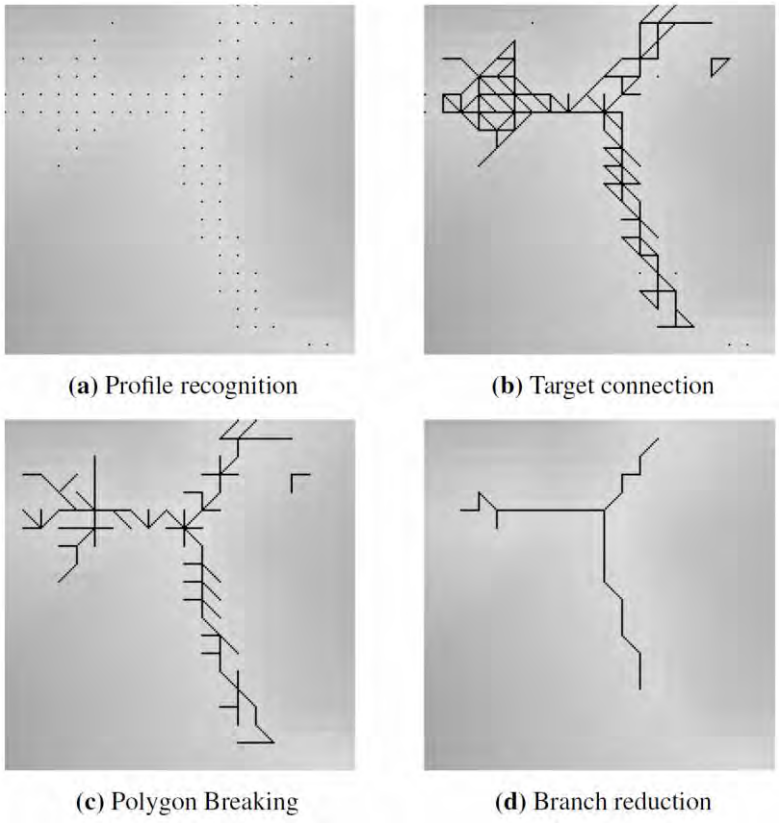


Figure 4.2: Different steps of ridge extraction with the Profile recognition and Polygon breaking Algorithm (Tasse et al., 2011). The final result is the minimum amount of points required to describe the main feature path.

1. Profile recognition is an algorithm that marks all points that could be part of a ridge line as potential candidates. To determine if a point is a candidate, the algorithm takes it as the centre of the profile. If there is at least one point with a lower height on both sides of the profile then it is marked as a candidate. Furthermore, the profile is switched from N–S, NE–SW, E–W to NW–SE to determine the candidacy of the point. Figure 4.2 (a) shows a series of points marked as candidates.

2. Target connection: All the adjacent candidates are now connected to form weighted segments, shown in Figure 4.2 (b). This process can produce many diagonal connections that

cross each other, when this occurs the program discards the less important edge. Edge importance is calculated by summing the height values of the two connecting points with a lower total weight being considered less important.

3. Polygon breaking: Target connection has the potential of producing closed polygons which need to be eliminated, as they can cause cycles in the resulting graph. The program repeatedly checks for closed polygons and removes the least important segment (lowest weight value) until there are no closed polygons of any size left (Figure 4.2 (c)). This process is achieved by taking all of the segments and sorting them by their weight. The PPA then checks the lowest segment to see if it is part of a polygon. If it is, the segment is deleted and the next lowest is checked. This process repeats until all of the segments have been processed. This produces a tree structure with edges lying along the terrain features.

4. Branch reduction: After Polygon breaking there are many short branches most of which are undesirable side effects of the Profile recognition stage generating too many redundant points. These short branches are repeatedly deleted a user-defined number of times. Figure 4.2 (d) shows the result of the Branch reduction step.

5. Line smoothing: This step moves the points to the average weighted position based on its location in relation to its neighbouring points. The weight of each point is valued proportional to its elevation for ridges and inversely proportional to elevation for valleys. The new position better fits the trend line and since it is an average it will never shift more than one grid space. The final output is a tree representation of the final ridge or valley feature lines (Figure 4.3(b)).

Tasse et al. (2011) were primarily concerned with performance and the extraction of large-scale terrain features. The original PPA was used with modifications to the Polygon breaking stage to make use of a *minimum spanning forest* algorithm while preserving the Profile recognition and Target connection steps. Kruskal's algorithm was chosen as it performs several orders of magnitude better than Polygon breaking. It is possible that the feature extraction data can be stored during a pre-process step and read in at runtime, saving valuable time, this is discussed further in section 5.4. Once the features have been extracted for both the DEM and user's sketch, the algorithm proceeds to the patch matching stage.

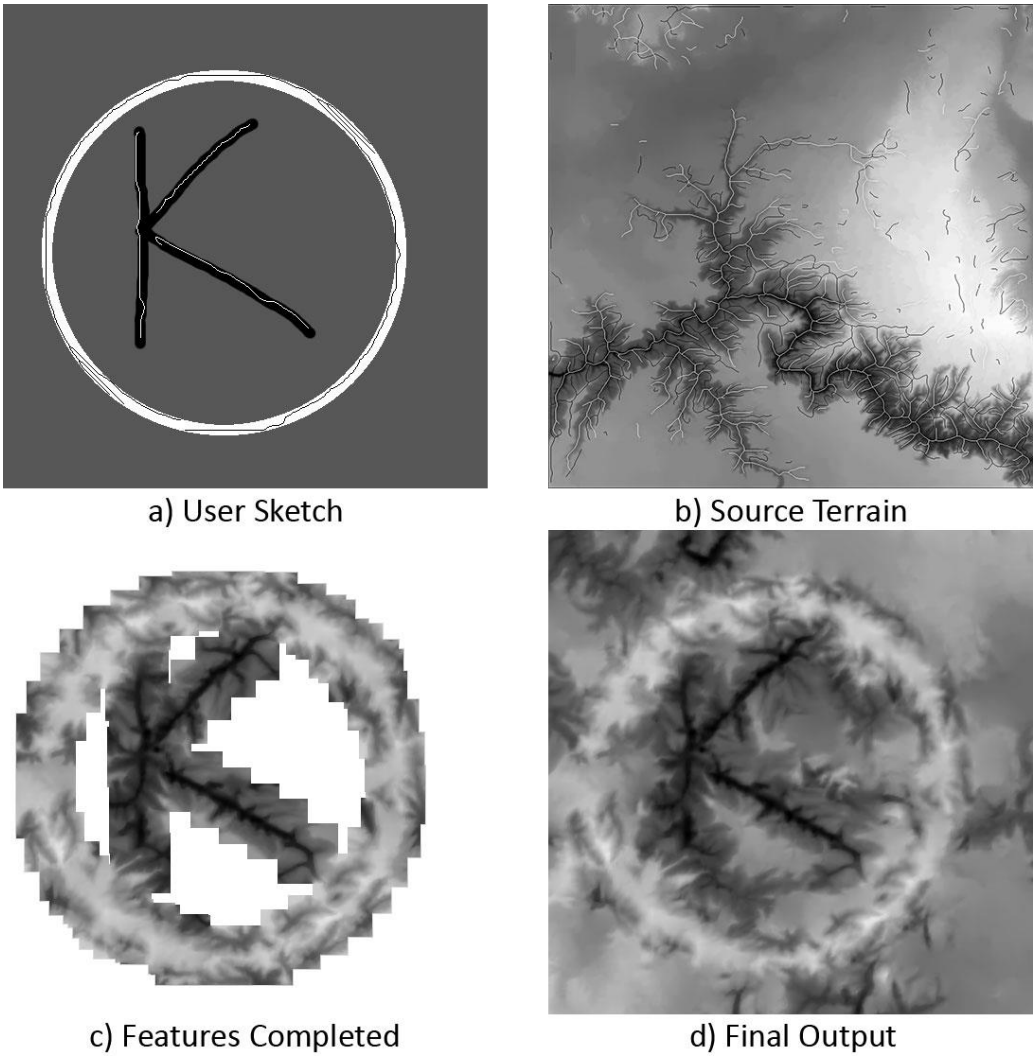


Figure 4.3: Patch-based texture synthesis. a) Users sketch input. b) Valley lines extracted from feature extraction on exemplar. c) Output after feature matching has completed. d) Final output after non-feature matching has completed.

4.2 Patch Matching

Tasse et al. (2011) make use of a single exemplar to provide the source data for the system. Feature extraction is run on both the exemplar and users sketch, with a collection of feature nodes. Patches are centred on these nodes locations, which encompass a detected feature. For the exemplar the patches are referred to as source candidates and are then rotated 8 times by 45° as well as mirrored along the x-axis and y-axis giving a total of 10 candidates per patch. These candidates are then compared to the patches from the user sketch. Patch matching is done in two stages: feature and non-feature matching.

4.2.1 Feature Matching

Feature matching compares patches from the set of source candidates against the set of user patches, searching for the best match. The output of the feature extraction process for both the source file and user input produces a tree data-structure with edges falling on the terrain features. These are connected with edge chains to form feature paths, as seen in Figure 4.2 (d). The matching process starts by constructing the candidate pool from these edges, by using each edge position as the central location of a candidate patch. The number of candidates is then expanded by rotating

and mirroring the patches, making the system more versatile. Feature matching follows a breadth-first traversal of the feature tree, preferably starting at the node with the highest degree of connectivity. The traversal proceeds along the paths in increments of one-half of the defined patch size, which ensures successive patches only partially overlap. At each node the control points are calculated, control points are the locations where the feature path intersects with a circle centred on the node with a radius, $r = \frac{PatchSize}{4}$. The number of control points describes the type of feature that this node is classified as. A single point indicates an end point; two points are a feature path and more than two indicating a branch point (Figure 4.4). They are used in a set of cost functions with the candidates to determine the best fitting patch. Three cost functions are evaluated; Feature dissimilarity (C_f), Angle differences (C_a) and Noise variance (C_n). These functions are multiplied by scalar values $(\alpha's)$ to control the level of influence each provides. The total cost (C_t) for candidate Q matching against target patch P is as follows:

$$C_{t(\mathrm{P,Q})} = \alpha_f C_{f(\mathrm{P,Q})} + \alpha_a C_{a(\mathrm{P,Q})} + \alpha_n C_{n(\mathrm{P,Q})}$$

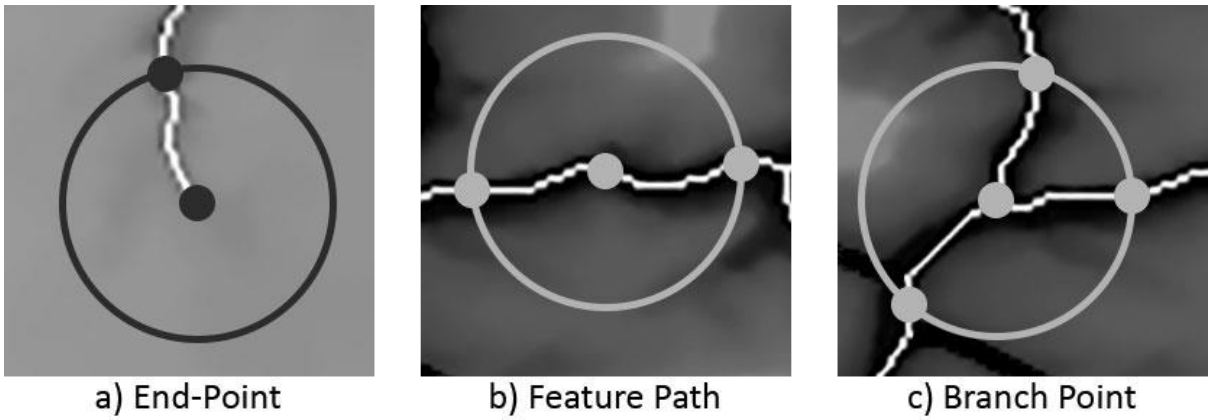


Figure 4.4: Example of different feature types based on the number of control points. a) Feature end point. b) Feature path. c) Feature branch.

Feature dissimilarity

This function determines the similarity between the user and candidate patch by comparing their height profiles using an L_2 norm. The height profiles consist of the height values along the outgoing feature path, both along the feature path and perpendicular to it. These paths are shown in Figure 4.5 (a and b) and represented by lines O and P, respectively. Candidates with a lower feature dissimilarity cost are more suitable matches.

Angle differences

The angles of the paths between the nodes for the given user and candidate patches are compared using a normalized sum of squared differences. This angle difference indicates how similar the structure of the candidate is to the user patch in terms of the top-down direction of the feature.

Noise variance

The noise variances of the user and candidate patches are computed at multiple levels of resolution and their sum-of-squared differences added to make up this cost component. The noise variance for given patch (P_l) for levels $l > 0$, with $l = 0$ being the coarsest level, is the variance of Gaussian noise computed by consecutively downsampling and upsampling P_l to obtain the lower resolution P_{l+1} and subtracting P_{l+1} from P_l. This process produces a set of frequency bands where

the details have been smoothed out. This cost function compares the noise differences between the two patches at different frequency band levels. Lower variance between the patches indicates that they have similar characteristics in terms of bumpiness at both coarse and fine scales.

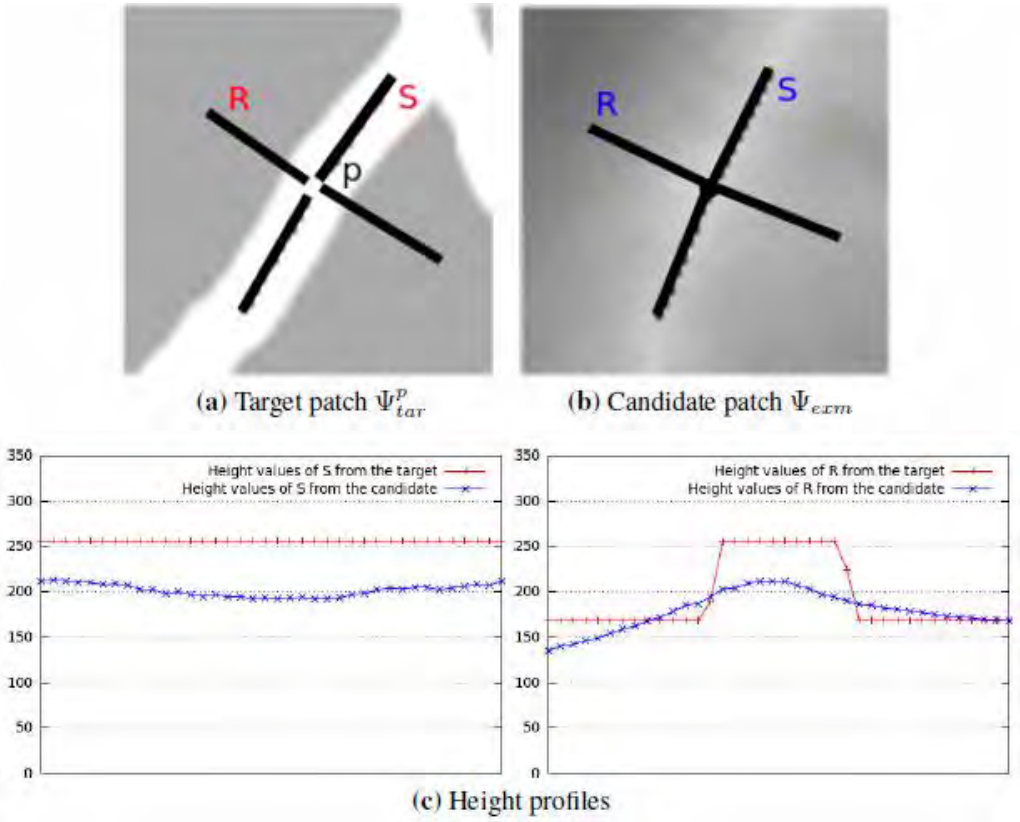


Figure 4.5: Feature dissimilarity Tasse et al. (2011), an illustration of how the algorithm examines the pixel data in a patch. (a) User patch. (b) Candidate patch. (c) Height profile for values perpendicular to path. (d) Height profile for values along path.

After all of the candidates are evaluated the total costs are sorted in increasing order, from this the first five (lowest cost) candidates are selected. These final candidates are now run through the Graph-cut cost algorithm used during merging (section 4.3.1). This algorithm evaluates the suitability of the optimal seam. A high cost indicates a greater impact on merging due to more dissimilar pixel data. The candidate with the lowest cost is selected as the matched patch and sent for merging into the output terrain as described in section 4.3. This process repeats for all the user patches resulting in the output shown in Figure 4.3(c). The system then starts matching the non-features to fill in the missing regions.

4.2.2 Non-Feature Matching

After the feature matching is completed the output terrain may contain areas where no data has been populated. To fill these *holes* with data, non-feature matching is performed. For this process, new candidate patches are created from areas in the exemplar that contain no significant feature data. These candidates are matched against already synthesised data in the output terrain and fill in the empty region Φ. Criminisi et al. (2004) show that the quality of the output terrain is affected by the order of the filling process and propose a filling algorithm that prioritises patches along structures. Tasse et al. (2011) made use of a similar algorithm for their implementation of the non-feature synthesis, which ensures terrain features are preserved and propagated correctly.

Patch-based filling algorithm

This algorithm determines the location of the next patch to undergo matching and is based on a best-first filling approach. The selection depends on priority values that are associated with every point lying on the boundary of Φ, $\partial\Phi$. For a given patch P_n, where $n \in \partial\Phi$ (Figure 4.6), its priority value is influenced by a *confidence* and *data* term. The confidence can be described as a measure of the amount of reliable information surrounding the pixel n. It is calculated by summing the number of pixels in the patch area that contain already placed data, then dividing by the total number of pixels (PatchSize^2). This gives a representation for the amount of already placed data the patch contains. The data term is a function of the strength of the isophotes (linear structures) hitting the front of the boundary of Φ, where there is no valid data. This term increases the priority for patches that an isophote flows into. This is fundamentally important to the algorithm because it encourages linear structures to be synthesised first and propagate securely into the output terrain. These two terms are combined together to yield the priority value for P_n.

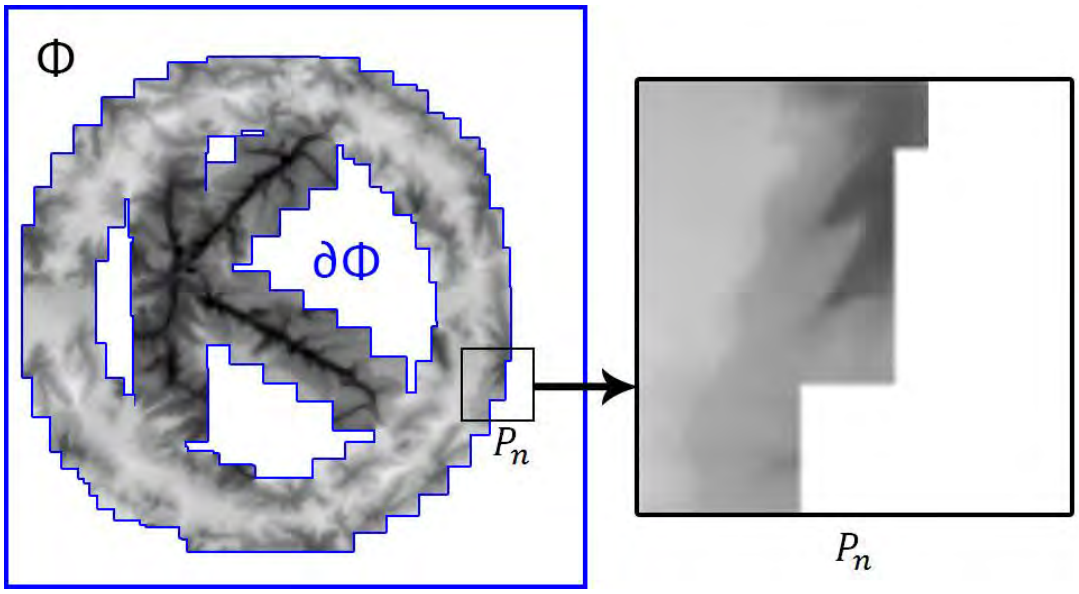


Figure 4.6: Example showing the empty region Φ, with the boundary $\partial\Phi$ highlighted in blue. A patch P_n centred around a point on $\partial\Phi$ is enlarged.

Criminisi et al. (2004) proposed a filling priority that is calculated by multiplying the confidence and data terms. This, however, discards pixels with a data term equal to zero even if their confidence term is large. Instead, these terms can be added together as in Nie et al. (2006). At each iteration of the algorithm the pixel that has the highest priority value is selected. Once a new location has been determined, matching is used to select the best candidate to fill the area. Non-feature matching ends when there are no more empty regions to fill.

Matching process

The candidate patches do not contain any significant feature data and thus, the matching criteria are slightly different from that of feature matching. The set of candidates are evaluated against two cost functions: noise variance difference C_n and the normalized sum of squared differences over the overlapping area C_s. C_n minimises the difference in the bumpiness of the surface of the patch and C_s matches the pixel data already synthesised in the output terrain. The total cost C_t is computed by

adding the two cost values together after they are multiplied by a scaling factor α. This determines the amount of influence each function has on the output as follows:

$$C_t = \alpha_n C_n + \alpha_s C_s$$

Tasse et al. (2011) uses scaling values of $\alpha_n = 0.0001$ and $\alpha_s = 10$, which were chosen after observing the magnitudes of the individual components to ensure both contribute to the total weighting. After the candidates are evaluated, the costs are arranged in ascending order with the k best candidates selected for a second round of cost evaluations. These short-listed candidates are now evaluated against the Graph-cut equation to determine which patch would make the best fit. The best fitting patch is then selected and merged into the output. This process repeats until there are no holes left in the resulting terrain, marking the end of the terrain synthesis process (Figure 4.3(d)).

4.3 Patch Merging

Once the system has found a matching patch, it must be placed in the output terrain. If this new patch were simply pasted into the output and overlapped existing data, a seam would appear as if there are different pixel values in the new patch. Thus a system is required to seamlessly merge the new patch (P_n) with any existing data. A patch (P_o) is cut out from the target image at the location where the new patch is to be placed. The region where existing data in P_o overlaps with data in P_n is defined as Ω. Tasse et al. (2011) develop an improved merging algorithm using three different techniques: Graph-cut (Kwatra et al., 2003), Shepard Interpolation (Shepard, 1968) and a Poisson equation solver (Pérez et al., 2003). This new combination produces superior results to those of Zhou et al. (2007).

4.3.1 Graph-cut

Patch merging starts by performing a Graph-cut to determine the optimal seam between patches P_o and P_n over the overlap region Ω (Figure 4.7). Or more specifically, the minimum cost cut of the graph is required. This is a well-known graph problem; minimum cut (max flow) (Sedgewick, 2001) with easy to implement algorithms. Kwatra et al. (2003) use this algorithm with a weighted cost function M, which penalises seams traversing through low frequency variations. This function is used to determine the weight of the edge between pixels a and b in the overlap region and is defined as follows:

$$M(a, b, P_o, P_n) = \frac{|P_o(a) - P_n(a)| + |P_o(b) - P_n(b)|}{\left|G^d_{P_o}(a)\right| + \left|G^d_{P_o}(b)\right| + \left|G^d_{P_n}(a)\right| + \left|G^d_{P_n}(b)\right|}$$

with d representing the direction of the gradient, which is also the same as the direction of the edge ab. $G^d_{P_o}$ and $G^d_{P_n}$ are the gradients of the patches P_o and P_n along the direction d. The graph-cut process uses the optimal seam to select which data from the new patch P_n is to be placed into the output image. Figure 4.8 shows the main stages in the graph-cut algorithm. The optimal seam is now known but still visible in the target and further processing is required to hide it.

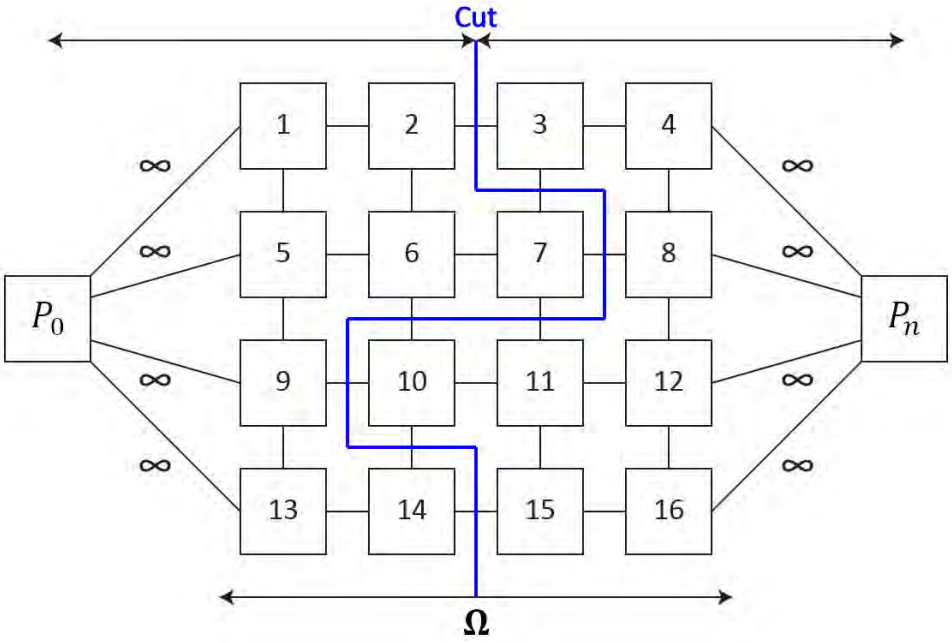


Figure 4.7: Illustration of the graph-cut algorithm between patches P_0 and P_n. The optimal seam connects adjacent pixels between the two patches.

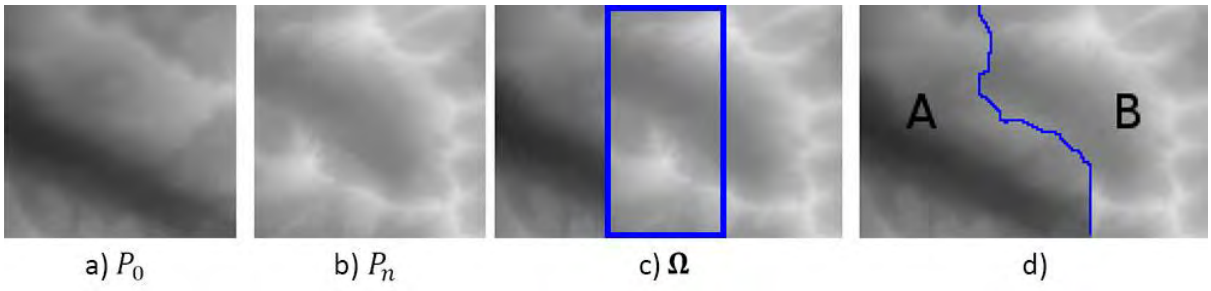


Figure 4.8: Example of the graph-cut algorithm steps. a) & b) Patches P_o and P_n. c) The overlap region Ω highlighted. d) The optimal seam between the two patches highlighted after merging.

4.3.2 Shepard Interpolation

A useful by-product of the Graph-cut process is that it partitions the target patch into two portions: the sink A containing pixels that come from P_o and the source B containing P_n. The visible seam can be removed by deforming B to match that of A along the cut. Tasse et al. (2011) chose to implement a deformation technique based on point features proposed by Milliron et al. (2002) to perform the deformation of the source producing B′. The pixels contained in B are displaced by $\Delta(x)$, which is calculated from $\mathrm{A}(x_i) - \mathrm{B}(x_i)$ for the points x_i along the seam and scaled by a distance-based normalised weight $\widehat{\omega}_i(x)$. The deformation is defined as:

$$\mathrm{B}'(x) = \mathrm{B}(x) + \Delta(x)$$

$$\Delta(x) = \left(\sum_{i=0}^{N} \widehat{\omega}_i(x)\right)\left(\mathrm{A}(x_i) - \mathrm{B}(x_i)\right)$$

with normalised weight:

$$\widehat{\omega}_i(x) = \frac{\omega_i(x)}{\sum_{j=0}^{N} \omega_j(x)}$$

where $\omega_i(x)$ is 1 at x_i and falls off radially to a distance d_θ. The weighting function ω_i was chosen to be an Inverse Weighting function as defined by Shepard (1968):

$$\omega_i(x) = \begin{cases} \left(\frac{d_\theta - d(x, x_i)}{d_\theta d(x, x_i)}\right)^\alpha, & if\ d(x, x_i) < d_\theta \\ 0, & otherwise \end{cases}$$

where $d(x, x_i)$ is the distance between x and x_i, d_θ the area of influence, and α the smoothness factor. This deformation process is better known as Shepard Interpolation with the results of this process following the graph-cut shown in Figure 4.9.

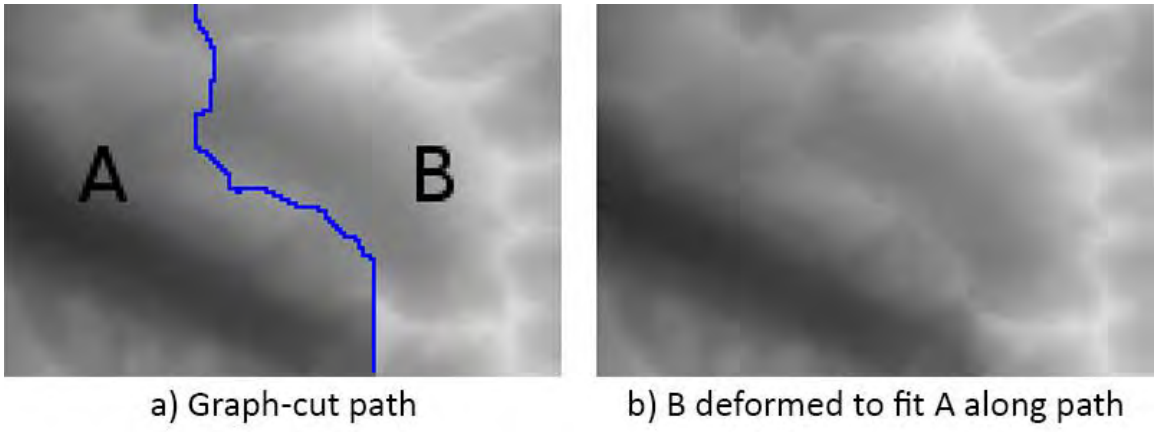


Figure 4.9: Results of Shepard Interpolation. a) Output from graph-cut algorithm. b) B is deformed to match the pixel values of A along the optimal seam.

A limitation of Shepard Interpolation is that it does not take into account the gradient values, which results in discontinuities found in the resulting gradient field. These discontinuities manifest in visual artefacts that are not visible in a 2D top-down representation of the terrain, but create obvious surface irregularities in 3D renderings of the terrain (Figure 4.11 (a-d)). This problem is solved by removing the optimal seam in the gradient field G_P instead of the height values in P. G_P consists of the gradient fields G_A from the sink A and G_B from the source B. The above equations are used by substituting $\mathrm{A}(x_i)$ with $G_\mathrm{A}(x_i)$ and $\mathrm{B}(x_i)$ with $G_\mathrm{B}(x_i)$.

$$G'_\mathrm{B}(x) = G_\mathrm{B}(x) + \Delta(x)$$
$$\Delta(x) = \left(\sum_{i=0}^{N} \widehat{\omega}_i(x)\right)\left(G_\mathrm{A}(x_i) - G_\mathrm{B}(x_i)\right)$$

The gradient field G_P is now free of discontinuities, since both G_A and G_B have the same values along the seam. The final step of patch merging is to calculate the new elevation values from the modified gradient field by solving a Poisson equation.

4.3.3 Poisson equation solver

The following Poisson equation with Dirichlet boundary conditions must be solved to calculate the final elevation values of patch P':

$$\nabla^2 P' = \nabla . G_P, \qquad P'|_{\partial\theta} = P|_{\partial\theta} \qquad 4.1$$

$$\nabla^2 P' = \frac{\partial^2 P'}{\partial x^2} + \frac{\partial^2 P'}{\partial y^2}$$

$$\nabla . G_P = \frac{\partial G_P^x}{\partial x} + \frac{\partial G_P^y}{\partial y}$$

$$G_P = \left(G_P^x, G_P^y\right)$$

where $\nabla^2 P'$ is the Laplacian of P', $\nabla . G_P$ is the divergence of the gradient field G_P and θ represents the entire patch area. Finite-difference methods (FDM) are used to construct a system of linear equations that approximate equation 4.1 (George, 1970). The Conjugate Gradient method of Shewchuk (1994) is used to solve the linear system for the unknown values of the new patch P'. The process starts by translating the height values within the overlapping area, which is approximately one-third of the patch size, into gradient values. Next the gradient values along the seam are set to zero. Now the Poisson equation is solved to determine the best set of height values to fit the modified gradients. The generation process is not confined to the boundaries of the patch, as neighbouring pixels may have been deformed during Shepard Interpolation, which also need to be considered. The newly generated values for P' are placed into the final terrain, which results in a smoothly transitioning terrain that is free of visual artefacts (Figure 4.11(e)).

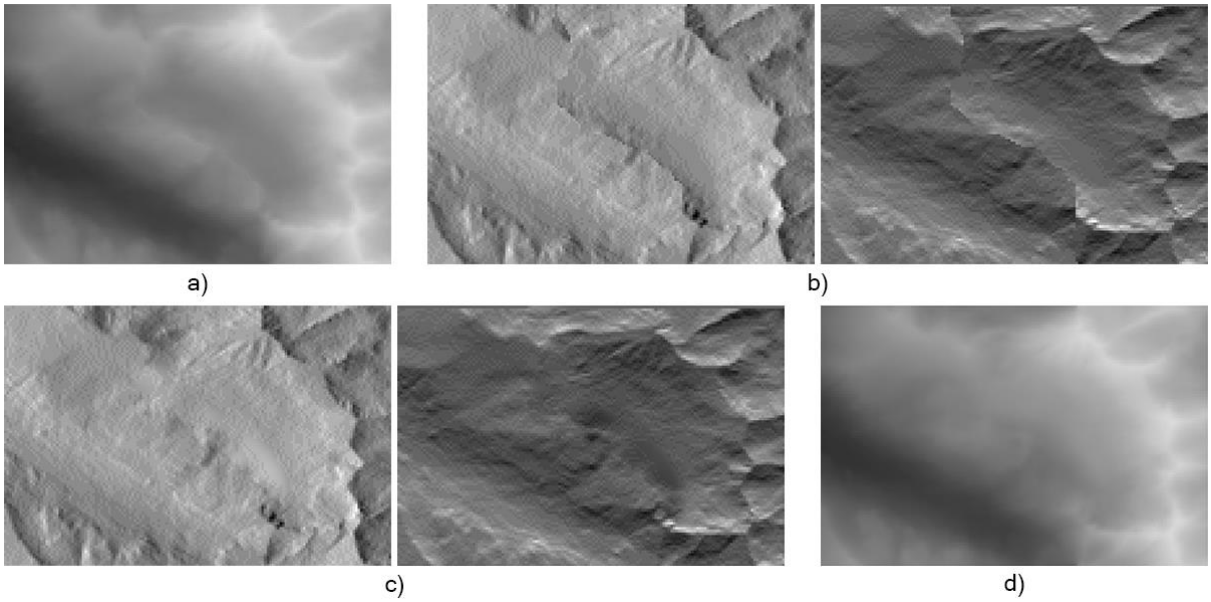


Figure 4.10: Poisson equation solving process. a) The image as output from Shepard Interpolation, patch P'. b) The gradient fields of the patch P'. c) The modified gradient fields free of discontinuities along the seam. d) The final output after the Poisson equations are solved.

4.4 Research Outcome

Tasse et al. (2011) improved on the already impressive results of Zhou et al. (2007) by: enhancing the matching process, increasing the candidate patches from the exemplar by a factor of 10; and the addition of noise variance along with some tweaks to the existing cost functions. A visual comparison of the results was conducted by Tasse et al. (2011) and found that in most situations their results produced better matches to the users input compared to previous work. They noted that the current approach to patch merging based on a combination of graph-cut and Poisson seam removal (Zhou et al., 2007), is not well suited to terrains. This is because the techniques produce terrains with discontinuities in the second order derivatives. These discontinuities appear as artefacts on the terrain and are more noticeable when viewing the output terrain in 3D. A user study was conducted which confirmed that their new patch merging technique, Shepard Interpolation with a Poisson equation solver, is superior and succeeds in eliminating the boundary artefacts (Figure 4.11). User experiments were conducted by Tasse et al. (2011) to determine the realism of their generated terrains. They found that there was no statistical significance proving that real unmodified terrains were superior to those generated by their system. They conclude that the realism of their terrain is not dissimilar to that of real-life landscapes. We identified several key aspects that Tasse et al. (2011) have improved over previous work:

- A terrain sketching interface allows for greater user control (Gain et al., 2009). Users will benefit from an interactive development environment with intuitive controls based on drawing $2\frac{1}{2}$D constraint curves.

- Performance issues of the PPA feature extractions are addressed using minimum spanning trees, similar to Bangay et al. (2010)'s work.

- By including noise variance similarity and sum-of-squared-difference cost functions on the overlapping region of already placed patches, the feature matching process has improved.

- By replacing the thin-plate spline deformation from the matching stage and mirroring and rotating candidate patches increases the sample pool. This increases the likelihood of finding good matches in the sample terrain.

- The feature dissimilarity cost function is modified to take height differences along outgoing branches of a feature into consideration.

- The filling order for non-feature matching is changed to a best-first filling approach based on gradient values (Criminisi et al., 2004). Noise variance similarity is also used during this phase.

- Lastly, a novel patch merging algorithm more appropriate to terrains is introduced. Along the optimal seam between two patches, discontinuities in the gradient field are removed with a scattered point interpolation and a Poisson equation solved for the new height values instead of just setting them to zero. This method produces a more realistic landscape, especially when viewed in 3D.

Tasse et al. (2011) propose possible extensions to their synthesis framework: utilising multiple DEM exemplar files and enhancing performance with GPU acceleration. In the next chapter we look into these extensions as well as some others in our enhanced framework.

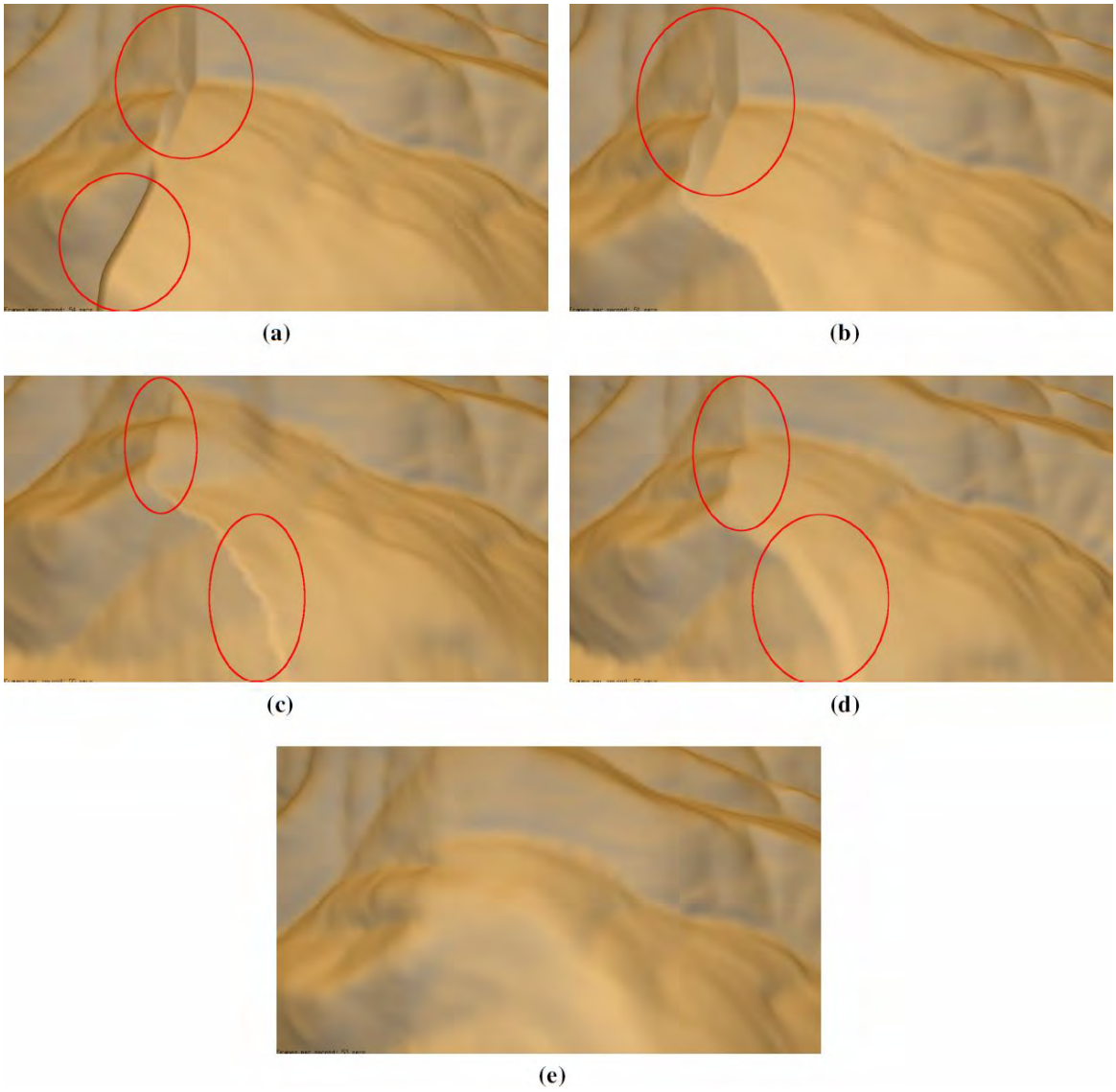


Figure 4.11: Comparison of patch merging techniques (Tasse et al., 2011). (a) No patch merging. (b) Graphcut algorithm. (c) Shepard Interpolation. (d) Results from Zhou et al. (2007). (e) Results from Tasse et al. (2011).

5 Enhanced Framework

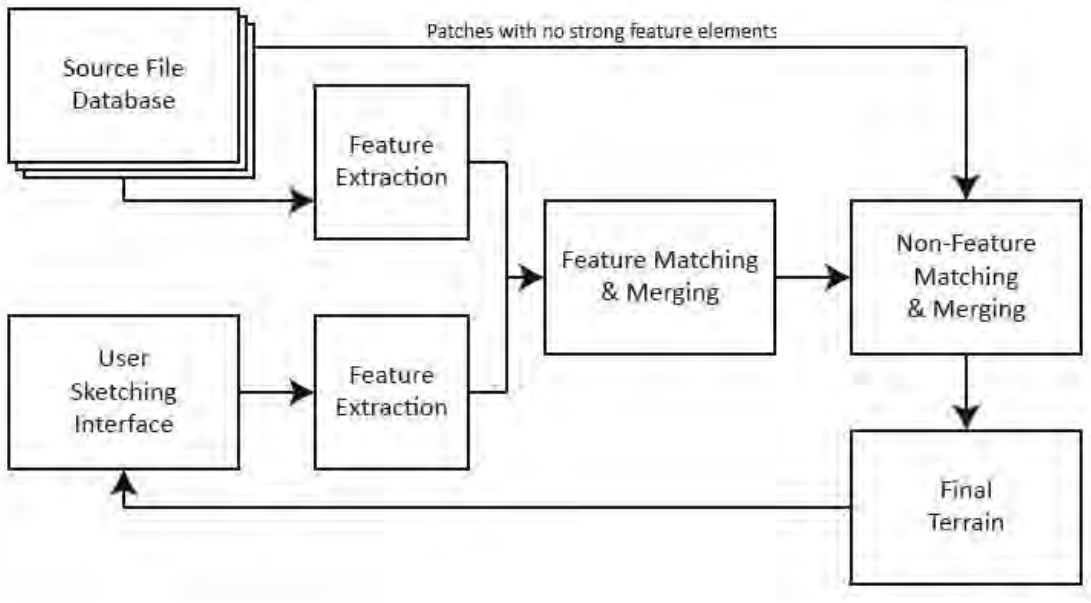


Figure 5.1: Overview of our proposed system for enhanced terrain synthesis. The entry-point to our system is the simplified sketching interface, which when synthesis is initiated, run through feature extraction to build the user candidates. A collection of varying source files is run through feature extraction also, with the feature data being used in matching and merging with the sketch data. A final step fills in the gaps left from feature synthesis with data from the source candidates to complete the terrain.

The terrain synthesis system of Tasse et al. (2011) produced compelling results due to their novel patch merging technique and enhanced user interface that allows 2½D manipulations of the terrain. However, some improvements can still be made to the system. An important improvement is support for increasing the candidate patch pool through the use of multiple input source files. This will allow for greater variability in synthesised terrains, provided it can be done efficiently. To accommodate this change, synthesis speed needs to be hugely increased. This requires parallelisation of core algorithms, extensive optimisation and careful use of pre-computation. We chose to use GPU-based parallelisation as the core of our extensions since a GPU is well suited to many image-based operations. Figure 5.1 is a design overview of our proposed system with the areas representing the main differences from the system of Tasse et al. (2011) outlined in red. This chapter describes the design of our framework so as to provide a high-level understanding of our objectives.

5.1 Multiple Input Sources

Previous work on texture synthesis mainly focuses on the use of a single source file, from which data is extracted. This limits the variation available, as it is highly unlikely that a single source contains enough variation to meet the user's requirements, particularly for highly varied or large terrains. For terrain generation this limitation is easily observed by using a single source file that contains mostly flat landscape data and attempting to synthesis large mountainous regions (Figure 5.2a). This can also lead to noticeable repetition during patch synthesis (Figure 5.2b). While we found no research relating specifically to terrain generation using multiple sources, Wei (2003) proposes a multi-source pixel-based texture synthesis technique. Their system minimises an error

function by examining all the pixel inputs within individual neighbourhoods (patches) to find the best set of input pixels. This error function is a weighted sum of the L_2 norm between the neighbourhoods of two different inputs. Unfortunately this technique does not adapt well to the system designed by Tasse et al. (2011). We found no other suitable research relating to the use of multiple input sources for terrain generation; instead we develop our own as an extension to Tasse et al. (2011).

We propose a system that supports a large number of input terrains in order to maximise the variation of data. This will allow for more diverse and rich landscapes to be generated. By using many individual files, the work can be more easily distributed to multiple processing elements without the need for complex logic to divide and distribute a single file amongst them. In order to provide this functionality, much of the underlying algorithm needs to be modified. During the feature and non-feature synthesis stages, the candidate patches from each input source are evaluated against the target patch. The best matching patches from each of the input sources are retained and compared with the overall best patch being selected for merging into the output terrain. The specifics of this process are discussed in detail in chapters 6 and 7. One disadvantage is that the all input sources will no long fit into memory: data must be streamed to and from the secondary memory. However, this is an acceptable cost given the improvements arising from a multiple source system. Nonetheless, in order to minimise the performance impact, additional optimisations will be introduced to speed up the overall processing pipeline.

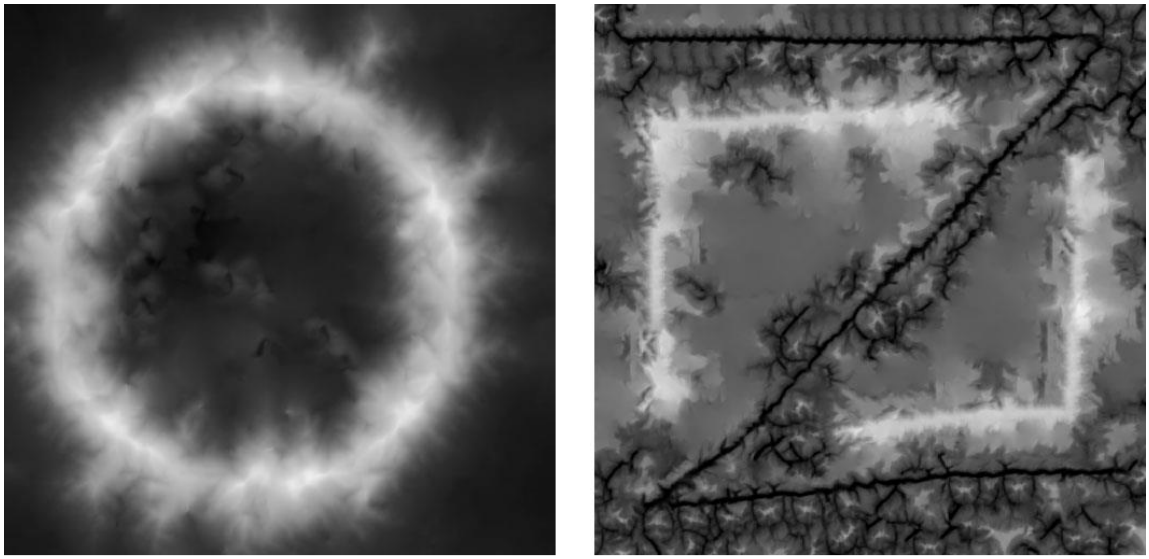


Figure 5.2: Examples of limitations with using a single source for terrain synthesis. (a) Using an input terrain without the correct type of feature data, source image lacks ridge details. (b) System can produce noticeable repetition in output terrain.

5.2 CPU and GPU Accelerated Synthesis

The process of synthesising terrains is complex and requires a large amount of processing power. The candidate patch searching algorithm is $O\big(a(m \times n)\big)$, where a is the number of source files for m candidates against the n user patches to match (Listing 5.1). Then there is the searching within each of the patches to evaluate the cost. Some aspects of this process can be parallelised since the algorithm is not dependant on each iteration's result. This allows for multithreading to be

implemented in order to exploit the parallelism of modern CPUs. For example, the looping over the n user patches for each candidate can be divided up amongst multiple processing cores. Assuming 4 processing cores, the complexity for the algorithm is reduced to approximately $O\left(\frac{a(m\times n)}{4}\right)$.

```
1  Loop over source files (a) {
2      Loop over candidates (m) {
3          Loop over user patches (n) {
4              Calculate cost of m for n
5          }
6      }
7  }
```

Listing 5.1: Algorithm overview for the candidate searching algorithm

However, some components of the system are sequential because they require information from previous iterations to complete. An important sequential component is the patch merging process. Merging selects the final candidate based on the information contained in the final output terrain. Despite these limitations, substantial performance can still be gained from a multithreaded approach. These gains can be magnified by leveraging the extremely high degree of parallelism that GPU devices offer. The details of these parallel implementations are covered in sections 6.3.2 and 6.4. By reducing the time required for terrain synthesis, the system becomes far more responsive and also allows larger terrains to be generated in reasonable time.

5.3 Simplified User Sketching Interface

To simplify the process of specifying the user's input to the system, a simplified sketching interface was designed. The interface provides the user with two pens that are used to draw the desired locations of ridges (mountains) or valleys, which make up the features. Once the user is satisfied with their sketch the synthesis option can be selected. The user is required to finish sketching all of their desired strokes as the system, while efficient, is not capable of interactive runtimes due to the large amount of data that requires processing. Next the drawn strokes are compressed down to produce a single image, which uses a binary system of black and white marking to represent the ridges and valleys. This image is run through feature extraction, which decomposes the sketched curves into a series of linked nodes/vertices in a graph structure. This graph structure is then split into a series of patches that are compared to the source candidates during synthesis. The details of the feature extraction process are described in section 6.1.

After all the features have been matched there may be many 'holes' in the output terrain where no features were drawn by the user. These areas are now filled with featureless data extracted from the input terrains in the form of patches. Once complete the final terrain is displayed in the interface. During the synthesis operation, several stages are rendered and kept as snapshots which can be viewed from the interface. These include the feature detection phase, which overlays thin lines over the sketch to show where the system detected features for both input terrains and user sketch. The result of each synthesis operation is also saved. The current output, representing the state of the system, is displayed in the interface and updated after every merge operation is completed. All these images can be easily switched between. The interface is updated continuously during the synthesis process so that the user can see each of the patches being placed into the output. The interface also allows the user to save the output terrain as well as the other snapshots as either a PNG image or Terragen terrain file. Loading of user sketches is also supported, which is

useful for testing purposes as it allows one to synthesise the same terrain multiple times for comparison purposes. Other useful tools include undo/redo commands and an eraser pen to make sketching easier. There are plans to extend the user interface further in the future as described in section 9.2. Figure 5.3 shows the different views of our interface.

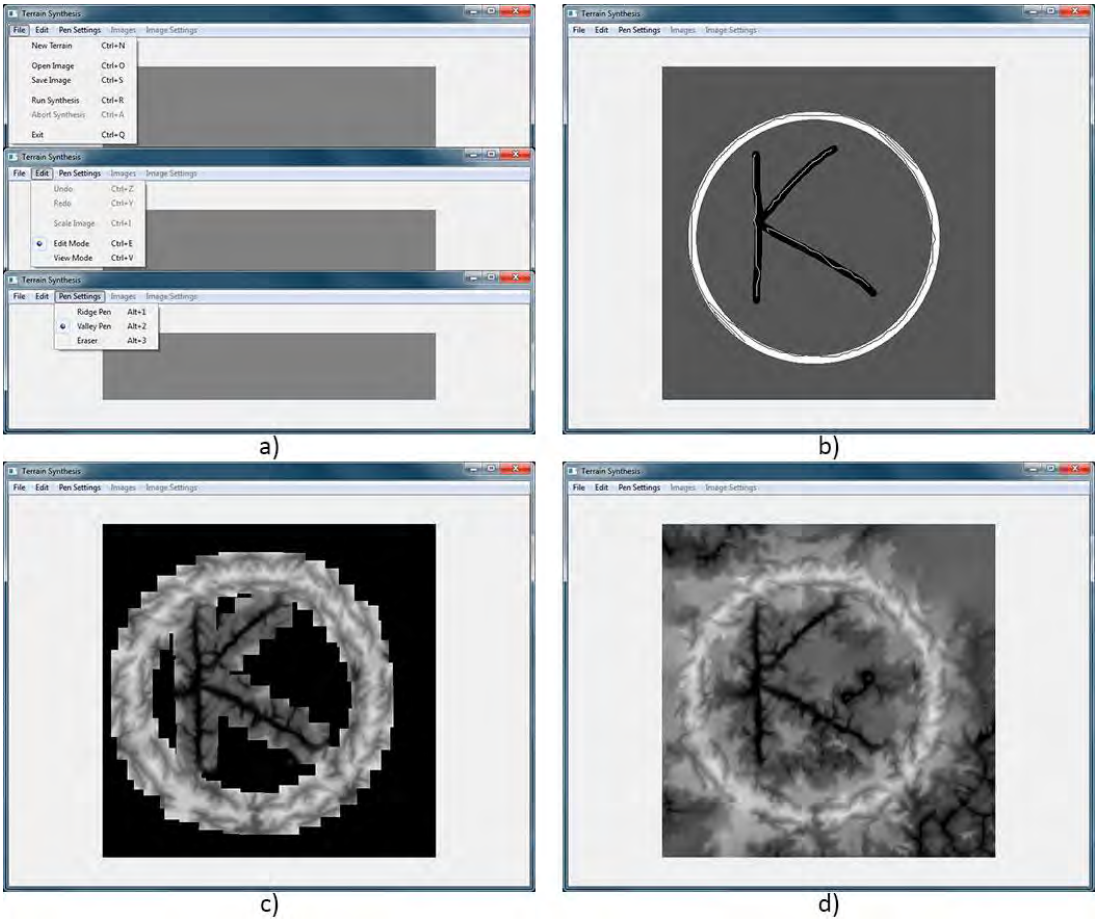


Figure 5.3: a) The main sketching interface with all menus expanded. b) Sample sketch drawn with feature detection run. c) Output after feature synthesis. d) Final output

5.4 Pre-Processors and Pre-Loaders

Running feature extraction on the input sources during synthesis wastes valuable time. The inefficiency can be addressed by pre-computing and storing the feature data and then reading it in when required. This observation brought about the notion of pre-processing data as an optimisation step. The feature extraction process (Section 4.1) is computationally expensive. The output of this phase is a graph of linked nodes (edge pairs) which represent x and y coordinates in the input file. These edge pair coordinates are saved out to an external ASCII file as newline delimited entries. By saving the data, each input file only has to be processed once. When loaded again these edge pairs reconstruct the graph to form a series of nodes, with each node being the central point for source patches. The patch size can be changed in the system to choose how much data around the node is used by the system. We present results for varying the patch size in section 8.3.3.

Pre-loaders are used to pull all the required data into memory in order to speed up processing. During the feature synthesis stage the raw image, along with the pre-processed feature extraction data for each source, is loaded into memory. If there is insufficient space for all sources, source data

is loaded in batches, with each batch being processed before the next batch is read in. Storing the data in memory allows for very quick access when calculating the best matching candidate for each of the user patches. The candidates for the source file are generated on-the-fly for processing and discarded when complete, thus limiting the total memory overhead by not keeping all candidates in memory constantly. This process is also batched by the system, if a particular source file has a very large number of candidates, to prevent it from running out of memory. This same principle is applied to the GPU implementations, with the source images and feature extraction data persisted in GPU memory. This is important as there is a very large overhead with transferring data from host to device, provided there is sufficient memory, otherwise batching is performed.

5.5 Summary

We presented an overview of our enhanced terrain synthesis framework that builds on the work by Tasse et al. (2011). It includes the use of multiple input sources to dramatically increase the candidate pool of data, which leads to better quality synthesised terrain. Through data pre-processing and optimisation of the synthesis pipeline we are able to reduce synthesis time despite the increase in input data. The next two chapters cover the implementation details for the synthesis process.

6 Feature Synthesis

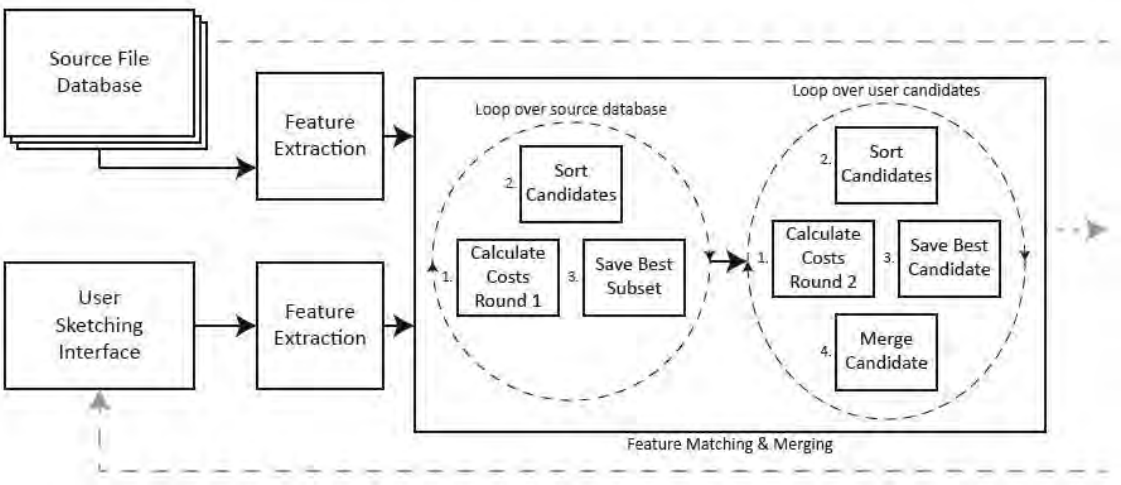


Figure 6.1: Feature synthesis pipeline showing flow of data for the Feature Matching & Merging block of our system (Figure 5.1)

Feature synthesis is the process of matching data from source terrains to a user sketch and placing them seamlessly into an output terrain. First the candidates are extracted along feature lines from the source terrain. These are then evaluated against the user patch by applying several cost functions to determine a subset of optimal matches. This process is extremely resource intensive and we present a parallel CPU and several different GPU implementations to speed up the matching process. Once all the source files have been processed, the overall best match is calculated with a second round of cost equations and then merged into the final image. The rest of this chapter explains the process in detail; Figure 6.1 shows an overview for the feature synthesis pipeline.

6.1 Feature Extraction & Pre-Loaders

Once the user has completed sketching their desired terrain they can start the synthesis process. To begin, the sketch they have drawn, is processed through feature extraction (Section 4.1), which generates the feature paths as a set of connected nodes. These nodes mark the central location for the user patches which are to be synthesised from the candidate data. The user's sketch is pre-processed to extract both ridge and valley data and then stored in memory. The control points, which describe the type of feature, are also calculated at this point for later use. The feature synthesis stage is then initiated. This stage requires the source files to be pre-loaded into memory for faster access subsequent to a pre-processing stage (Section 5.4). Feature synthesis is then run for all the user patches.

6.2 Cost Functions

The feature matching algorithm must quantify the difference between the source candidate and user patches. There are two rounds of cost calculations, with the first round occurring during the inner functions of feature matching (Section 6.3) and the second occurring before merging into the final image (Section 6.5). There are four cost functions in total: *Feature Profiling*, *Sum-of-Squared* differences, *Noise Variance* and *Graph-cut cost*. Feature profiling is part of round one; the others are

executed during round two. The reason for splitting this up is that the cost functions in round two make use of the already placed data in the output terrain, to provide better statistics.

6.2.1 Feature Profiling

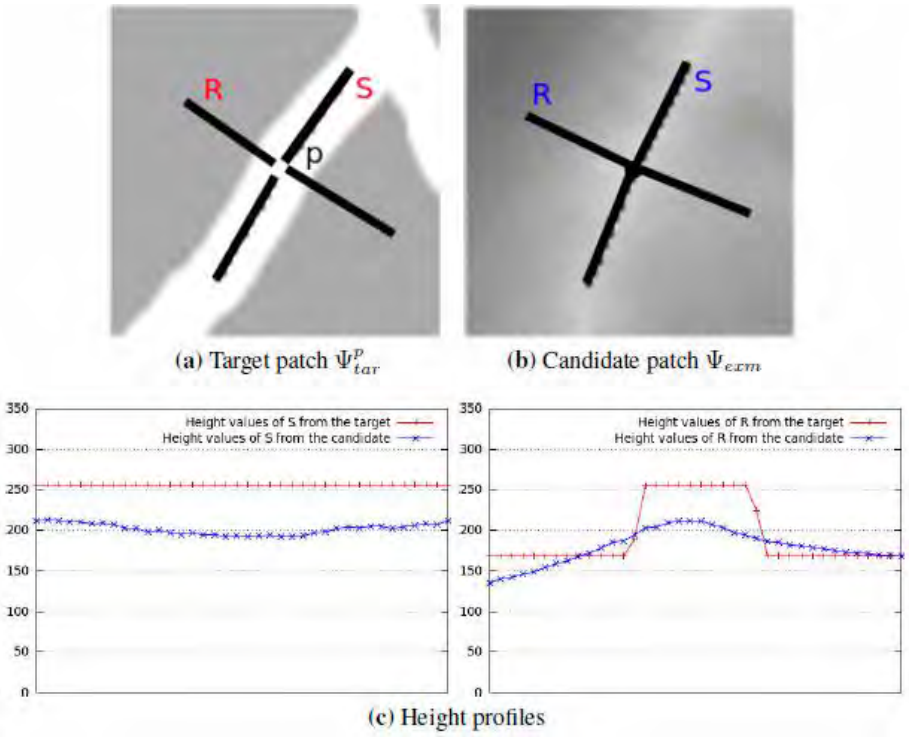


Figure 6.2: Feature profiling algorithm against user and source candidate patches. Segments r and s represent profile paths for the patches.

Feature profiling quantifies the similarity of features between the candidate and user patches. This cost is calculated by comparing the height profiles of lines r and s of the user patch ρ (Figure 6.2 (a) and (b)) with the L_2 norm (Sum-of-Squared Differences). Tasse et al. (2011) additionally compare the height profile along the feature path, in contrast to Zhou et al. (2007) who only compare height profiles perpendicular to the path. Figure 6.2 (c) illustrates the height differences between the target and source patches for both segments. The lower the calculated cost, the more likely the candidate is a good match for the user patch. This cost equation is used during the first round of evaluations as it only requires the feature path data and does not rely on already synthesised data. It therefore carries all the weight for the initial matching phase.

```
1   Inputs: User Candidate, User Control Points, Source Candidate
2   Output: Cost value (float)
3
4   Initialise: total = 0
5   Loop over control points {
6       Calculate points for segments r and s (Figure 6.2)
7       Run profiling for segments r and s {
8           Initialise: sum = 0, count = 0
9           Iterate over all points on segment {
10              Calculate difference: (Diff = Usr_pixel − Src_pixel)
11              Add difference squared: sum+= Diff^2
12              count ++
13          }
```

```
14          Add sum to total: total+= sqrt(sum)/count
15      }
16  }
17  Final cost: FP: total/#ControlPoints
```

Listing 6.1: Feature Profiling algorithm

6.2.2 Sum-of-Squared Differences (SSD)

The SSD function evaluates the differences between the source candidate and the user patches (Listing 6.2). Here, a lower overall cost indicates a better matching candidate patch. The user patch is extracted from the current output terrain, but as this may contain invalid (not yet synthesised data) this SSD is only evaluated across the overlapping area of the patches. The SSD evaluates the difference in the raw height values of the corresponding pixels, squares the difference and adds this to the sum for the candidate patch.

```
1   Inputs: User Candidate, Source Candidate
2   Output: Cost value (float)
3
4   Initialise: sum = 0, count = 0
5   for (i : PatchSize^2) {
6       Obtain user candidate pixel value (U_i)
7       if U_i is valid {
8           Obtain source candidate pixel value (S_i)
9           Calculate difference (Diff = U_i − S_i)
10          Add difference: sum+= Diff^2
11          count++
12      }
13  }
14  Final cost: SSD = sqrt(sum)/count
```

Listing 6.2: Sum-of-Squared Differences algorithm

6.2.3 Noise Variance

The noise variance for the source candidate and user patches is computed at multiple levels of resolution and the SSD of these differences is calculated. The lower the overall cost value the better the chance of the two patches having similar characteristics in terms of roughness of the terrain at different frequencies. We implemented the Wavelet Noise algorithm of Cook and DeRose (2005). The noise variance for an image at a given resolution level is the variance of the Gaussian noise produced by consecutively downsampling and upsampling the image, resulting in a lower frequency image which is then subtracted from the current level. For the purposes of our research we make use of three noise levels. A useful observation is that the orientation of the patches has no effect on the noise variance, meaning that it only needs to be calculated once for all orientation changes of the candidates. The cost calculation algorithm is presented in Listing 6.3. To limit the influence this cost value has on the overall candidate cost, it is scaled by $\alpha = 0.001$. This balances out the cost so as not to overpower the SSD cost value.

```
1   Input: Noise variance arrays (User and Source candidates)
2   Output: Cost value (float)
3
4   Initialise: sum = 0
5   for (i : #Levels – number of levels of generated noise variance) {
6       sum+= (UsrVar[i] − SrcVar[i])^2
```

```
}
Final cost: NV = sqrt(sum) / #Levels
```

Listing 6.3: Noise Variance algorithm

6.2.4 Graph-cut cost

During patch merging (Section 6.5) a Graph-cut (Section 4.3.1) is performed between the output terrain and the best overall candidate patch, which determines the optimal seam between the two. As part of this process, the minimum cut (max flow) is calculated in order to find the optimal seam along which to cut. We use this value to quantify the placement of the candidate patch in the output terrain (Listing 6.4). There is a large overhead in running this calculation and it is thus only executed on a small subset

of the five lowest cost candidates with the best patch being selected for the merging process.

```
Inputs: Destination patch from output terrain, Candidate patch
Output: Cost value (float)

Initialise graph-cut algorithm {
    Create vector to store cuts (C)
    Loop over pixels in patch (PatchSize^2) {
        If pixel is valid then add coord to C
    }
    Loop over C {
        Determine sink / source status of cuts coord
    }
}
Initialise graph structure (G)
Loop over C {
    Calculate weight and add as edge to G
}
Calculate maximum flow of G and return as cost value
```

Listing 6.4: Graph-cut cost algorithm

6.3 Feature Matching – CPU

Our first implementation of the feature matching algorithm is a sequential CPU algorithm. We develop two different versions of this algorithm with the difference lying in the order of looping over the datasets. We then select the more efficient of the two in terms of both speed and memory efficiency and develop a parallel implementation to further improve performance. During terrain synthesis the feature matching process is executed twice, once for matching ridges and once for valleys. In this section we first discuss the cost calculation process and then present the sequential and parallel implementations.

6.3.1 Sequential CPU Implementation

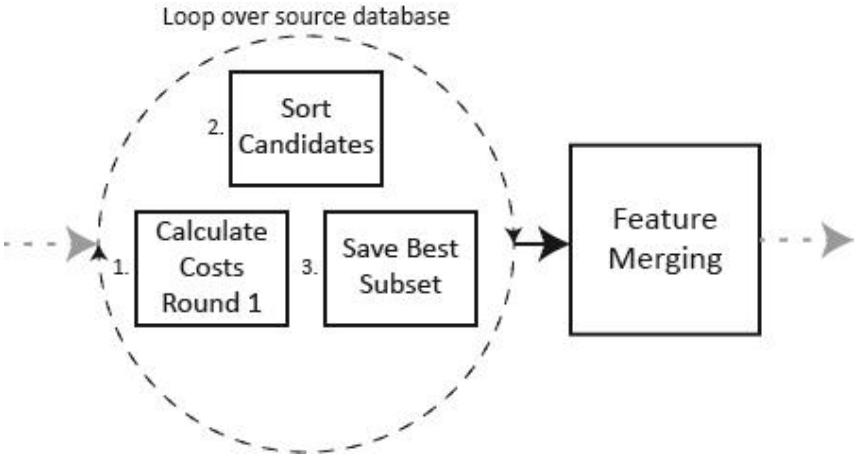


Figure 6.3: Overview of the second version of sequential CPU feature matching. Feature merging is included as it is a required part of the flow. More information on the merging process is found in section 6.5.

The design of the sequential implementation is adapted from Tasse et al. (2011) and extended to account for multiple input sources. An overview of this process is provided in Listing 6.5 representing version one of the sequential implementations. It starts with the data obtained from feature extraction of the user's sketch, which is processed to produce a set of user nodes. These nodes represent ridge/valley features that need to be matched against data in the source terrains. The algorithm loops over these nodes; this is done so that data is placed into the final output sequentially and then used in cost calculations for successive patches. Within each iteration, the current user node is prepared: the pixel data from the users sketch is extracted centred on the node. The control points calculated during feature extraction are also obtained, which describe the type of feature this node is. This data represents the user patch that is to be matched against the source files and is run as an inner loop.

```
1  Inputs: User Sketch, Source Files, Source Feature Extraction Data
2  Output: Image with features synthesised
3
4  Extract features from user sketch
5  Verify features were found (abort if none)
6  Prepare user nodes
7  Loop over user nodes {
8      Prepare user patch
9      Loop over source files {
10         Verify features
11         Prepare source nodes
12         Prepare source patches (candidates)
13         Calculate costs for all candidates
14         Sort candidates on cost
15         Save subset (best-set)
16     }
17     Find best overall patch from best-sets
18     Merge best patch
19 }
```

Listing 6.5: Algorithm overview for the version one of sequential feature matching.

All the source files are evaluated in this inner loop (source-loop), with the first step being to ensure the source file has been pre-processed and pre-loaded into memory. The pre-processing step

runs the feature extraction algorithm in the background; the results are saved to hard disk to prevent the need to run this costly process repeatedly. There is a small chance this has not happened yet if there are a large number of source files in the system. The process will block until the data is properly available, as the pre-processing is carried out in a separate thread (see section 6.6 on optimisations). The source file now undergoes a preparation stage, where the raw extracted feature data is processed into nodes. This state verifies that this file has the correct type of features (ridge / valley) before converting these nodes into patches. During the conversion process the patches undergo a series of transformations. They are rotated eight times through 45° increments as well as mirrored along the x and y axes, to produce ten variants for each patch (hereafter referred to as the candidates). At this point the candidates are matched against the user patch according to the feature profile cost function (Section 6.2.1), to produce the cost values for each candidate. This data is now sorted in ascending order and a small subset of the best candidates (best-set) is stored for additional processing. For the purposes of our research we use a subset size of five candidates per source file, because we found that larger set sizes increased synthesis time due to the more complex second round of cost calculations. Smaller subset sizes would reduce the variability at merge time, which could potentially lead to the same patch being placed adjacent to one another. This would create noticeable visual artefacts, which we can address by skipping the selection of the same patch for adjacent merge operations. The matching process continues evaluating all source files and adding the best-set of each to an array. Once this array is complete it is returned to the user-loop where processing continues.

```
1  Inputs: Array of best candidates
2  Output: Single best patch
3
4  Initialise variables
5  Extract user patch (U) from current state of final output terrain
6  Generate noise statistics for U
7  Loop over best candidates array {
8      Extract candidate (C) from its source file
9      Run sum-of-squared differences cost function for C against U
10     Run noise variance cost function for C against U
11     Run feature profiling cost function for C against U
12     Record sum of cost functions for candidate
13 }
14 Sort the candidates cost value in ascending order
15 Loop over best five {
16     Run graph-cut cost function for C against U
17 }
18 Select lowest cost patch as best overall
```

Listing 6.6: Algorithm overview for selecting the best overall patch

After all source files have been evaluated there is an array of best candidates with a maximum size of $(\#SourceFiles \times 5)$, which is processed to determine the best overall patch for merging. There can be fewer candidates if there is insufficient feature data in a source file to provide a reasonable match. At this point a second round of cost equations is run on all these candidates to find the best overall patch, as described by Listing 6.6. There are two parts to this. Firstly all the candidates are evaluated with the SSD, noise variance and feature profiling cost functions. They are then sorted again and the best five candidates are run with the Graph-cut cost equation to work out the best patch to merge. This best patch is now merged into the final output terrain as described in Section

6.5. The process repeats until all the user patches have been matched, at which point feature synthesis is complete and non-feature synthesis is executed to fill in any gaps in the output terrain with data from the source files that contain no strong features (featureless). This process is described in chapter 7.

The implementation above is optimised for use with a single source file as per Tasse et al. (2011), because the source candidates and user patches can be directly compared to each other while being kept in memory for fast access. By extending it to support multiple input files, each of these sources needs to be loaded into memory and candidates extracted for every user patch. This requires the source files to be swapped in and out of memory multiple times leading to a large performance overhead. Another approach is to retain the user patches in memory and allow for the source files to be loaded once per synthesis and compared against all user patches. This solution trades space efficiency for time efficiency. Results for the differences between version one and the optimised version two are presented in section 8.1.1.

The alternative version of sequential feature matching (Listing 6.5) is described in Listing 6.7. The process starts by processing the user's sketch through feature extraction and generation of the nodes as before. At this point all the nodes are processed to create an array of user patches. This array is cached in memory and has provision for storing the best-set candidates for all the source files. This requires a larger amount of memory as the best-set for all source files and all user patches' needs to be stored. This is because the merging process can only take place after all the source files have been evaluated.

```
1  Inputs: User Sketch, Source Files, Source Feature Extraction Data
2  Output: Image with features synthesised
3
4  Extract features from user sketch
5  Verify features were found (abort if none)
6  Prepare user nodes
7  Create array of user patches
8  Loop over source files {
9      Verify features
10     Prepare source nodes
11     Prepare source patches (candidates)
12     Loop over user patches {
13         Calculate costs for all candidates
14         Sort candidates on cost
15         Save subset (best-set)
16     }
17 }
18 Loop over user patches {
19     Find best overall patch from best-sets
20     Merge best patch
21 }
```

Listing 6.7: Algorithm overview for the version two of sequential feature matching.

Next the system loops over all the source files, verifies features, prepares the nodes and generates the candidates. All the user patches are then evaluated against all the candidates with the best-set of candidates for each user patch stored for later processing during the merging loop. After all sources have been processed the system finally loops over the user patches to process the best-set candidates for each. The overall best patch is determined through a second round of cost

calculations and then merged into the final terrain, with feature synthesis concluding after all user patches have been merged. Based on the results (Section 8.1.1), we observed a marginal speed improvement over version one. This improvement comes from not having to swap the source files, which includes a preparation step before cost calculation. It is mostly mitigated from our pre-processing and caching optimisation. However, this version scales better when a larger number of source files are used and provides an easier implementation for distributed computation, as there is less context switching of data involved. It was decided that this would be the optimal solution and subject to further optimised through multithreading as described in the next section.

6.3.2 Parallel CPU Implementation

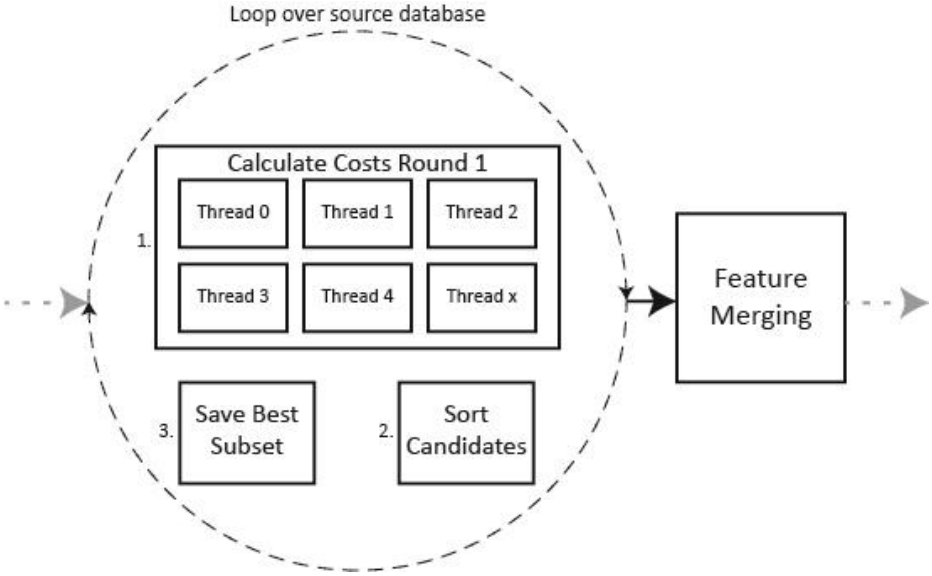


Figure 6.4: Overview for parallel CPU feature matching.

The parallel CPU implementation builds on Sequential CPU version two by adding additional threads to several of the components to better utilise the multiple cores of modern CPUs. Figure 6.4 shows the same version two pipeline but with the parallelised parts highlighted. There are two components that we have targeted for multithreading: the pre-processing stage and calculating the candidate costs for both rounds. In order to support the use of multiple threads, some changes were required to manage the threads and correctly distribute the workload. The pre-loader system distributes the workload by giving each thread a different source file to process. This works well as there is consequently no shared data between the threads.

During the matching process a large number of candidates are compared to the current user patch. This process can benefit greatly from multithreading with the candidates being equally divided between the threads. Once all the candidates have been processed by all the threads, a single thread then performs the sorting process with the best-set stored before moving onto the next user patch. Once all the user patches have been evaluated then the next source file is processed. The parallelisation in the system could be extended by unrolling one of the loops. Then having each of the source files processed by a separate thread controller, which itself spawns multiple threads for the cost calculations. However, this would not improve performance and could actually hamper it – CPUs have a low number of cores and creating too many threads would force

the CPU to thread swap constantly resulting in lower overall performance. This idea, however, is explored in section 6.4 with GPU matching.

Once all the source files have been processed, the system starts the merging process with the second round of cost calculations. Again, the cost calculations for the candidates are divided up between multiple threads to speed up processing. A single thread is required to perform the sorting and the final merging process before moving onto the next user patch.

6.4 Feature Matching – GPU

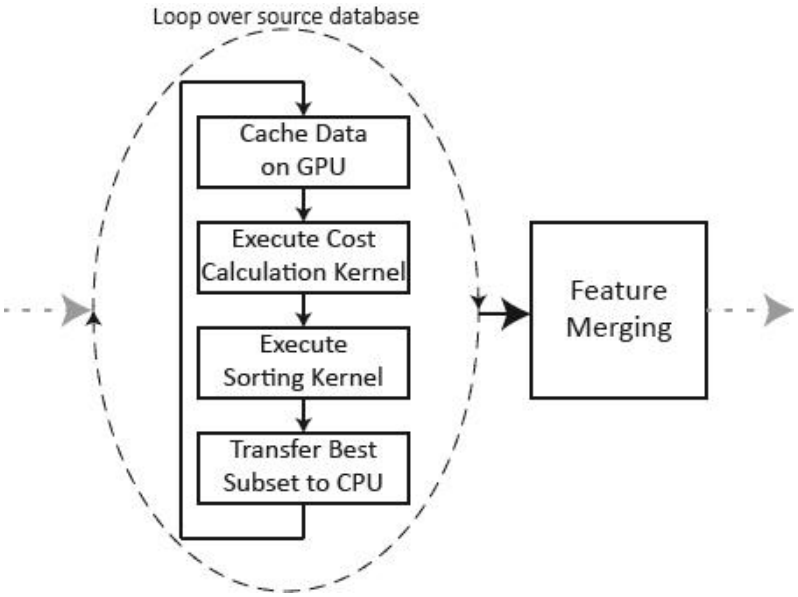


Figure 6.5: Overview of the GPU feature matching pipeline

This section discusses our implementation of feature matching on GPU devices to further reduce the synthesis time by exploiting the high degree of parallelism the GPU offers. Figure 6.5 shows an overview of the GPU pipeline with the components processed on the CPU and GPU as shown. In order to maximise the performance, a good balance between host (CPU) and device (GPU) is required. We show how the design of our parallel CPU implementation can be adapted so that the most suitable parallelisable components can be processed on the GPU.

6.4.1 Caching of data on GPU

Before any processing can be performed on the GPU, the data needs to be transferred over from host memory to the devices global memory. There is a very high overhead in transferring such data, so it is desirable to pre-load the data and cache it on the GPU. The first time a synthesis operation is begun, the data is transferred, which means that running another synthesis will be faster as there is no transfer cost. For the purpose of testing in Chapter 8, we run all tests after the data has been cached. Once the data is cached in the devices global memory it is transferred to higher speed device memory for cost calculation. We implemented several CUDA kernels to compare the use of these different types of memory (Section 6.4.3).

6.4.2 User Patch Extraction

The first stage of running the synthesis on the GPU is to process the user's sketch through feature extraction. This is still performed on the CPU, but the user node data is transferred to the GPU. The user patches are now extracted from the data according to the pseudo code in Listing 6.8.

```
1  Inputs: User Patches
2  Output: None (Results kept in GPU memory)
3
4  Allocate memory on GPU {
5      Patch cords: 2 × (#UserPatches × SizeOf(int))
6      Patch control points: 2 × (#UserPatches × #MaxControlPoints × SizeOf(int))
7      Number of Patch control points: 1 × (#UserPatches × SizeOf(int))
8  }
9  Copy user data from host to device
10 Allocate memory to store user patches: (#UserPatches × PatchSize² × SizeOf(float))
11 Init kernel: GridDim = (PatchSize/32, PatchSize/32, #UserPatches) : BlockDim = (32,32)
12 Execute extraction kernel {
13     Calculate lookup coordinates for sketch
14     Calculate memory address for output
15     Extract pixel data from source
16     Store data in user patches memory
17 }
```

Listing 6.8: Overview for the user patch extraction on the GPU

Firstly, blocks of memory are allocated on the device, and the user's sketch and accompanying feature extraction data are loaded into this space. An additional large array is allocated, which stores patches extracted from the user sketch. The user patch extraction kernel is then executed, using a grid with dimensions of $\left(\frac{PatchSize}{32}, \frac{PatchSize}{32}, \#UserPatches\right)$ and block dimensions $(32, 32)$. The blocks are sized to fit within a single CUDA warp to maximise efficiency. We divide up the grid so that multiple blocks can process a single patch and add a 3rd dimension to iterate over all the user patches. One pre-requisite is that we select a patch size that is a multiple of the CUDA warp size – for our research we used a standard patch size of 64. Zhou et al. (2007) made use of a patch size of 80 pixels, they determined this by the spatial scale of the source file being used as well as the detail of the resulting image required by the user. Tasse et al. (2011) also make use of the 80 pixel patch size, we chose to use a smaller patch to better suit our GPU system design and although marginally smaller it should not impact the output much. We do a comparison of our system with that of Tasse et al. (2011) in section 8.3.1, where we provide the output images from their system with ours using a patch size of 80 and then of 64. After the user patch extraction kernel is executed, we are ready to process the candidates.

6.4.3 Candidate Cost Calculations

The system now loops over the source files to analyse the costs of all the candidates against the user patches. The process starts by allocating memory on the GPU to store the source image along with the data obtained from feature extraction. This data is only cached to the GPU once, at first runtime, for every source file. The largest computation time comes from calculating the costs of all the candidates for all the user patches. We develop a total of eight different GPU versions to investigate various techniques to maximise performance.

Initial GPU versions

The first four versions share a similar starting point, memory is allocated to store a single candidate patch $(PatchSize^2 \times SizeOf(float))\ bytes$. There are two nested loops that are run, the first loops over all the source patches and the second over the number of candidate transformations, this is set at ten variations per patch. Inside the inner loop the candidate image data is extracted. A kernel with grid dimensions $\left(\frac{PatchSize}{32}, \frac{PatchSize}{32}\right)$ and block dimensions $(32, 32)$ is used to efficiently extract the candidate patch from the source image. When running the cost calculation, there is an additional loop which iterates over all the control points for the user patch. This number describes the kind of feature that was detected during feature extraction and is thus variable for each patch. The code then diverges into one of the four different versions, although with each setting the grid dimension equal to the number of user patches to process.

```
1   Inputs: Source File, Patch Offset
2   Output: Candidate patch stored in GPU memory
3
4   Init Kernel: GridDim = (PatchSize/32, PatchSize/32) : BlockDim = (32,32)
5   Execute extraction kernel {
6       Calculate lookup coordinates for sketch
7       Calculate memory address for output
8       Extract pixel data from source
9       Store data in user patches memory
10  }
```

Listing 6.9: Overview for the candidate patch extraction kernel

The first version (Listing 6.10) is a simple attempt that uses a block dimension (1) that makes use of a single thread and executes all the code sequentially. This is highly inefficient and the performance results reflect this (Section 8.1.3), although still much faster than a sequential CPU implementation. The goal of this implementation was to directly translate our CPU implementation and provide a base to work from for GPU optimisations.

```
1   Inputs: Source Candidates
2   Output: Calculated costs for candidates – Stored in GPU memory
3
4   Allocate memory on GPU {
5       Candidate Patch: PatchSize × PatchSize × SizeOf(float)
6   }
7   Loop over source patches {
8       Loop over number of source transformations {
9           Execute candidate patch extraction kernel (Listing 6.9)
10          Init kernel: GridDim = (#UserPatches) : GridBlock = (1)
11          Execute cost calculation kernel {
12              Loop over control points {
13                  Calculate points for segments r and s (Figure 6.2)
14                  Run profiling for segments r and s {
15                      Initialise: sum = 0, count = 0
16                      Iterate over all points on segment {
17                          Calculate difference: (Diff = Usr_pixel − Src_pixel)
18                          Add difference squared: sum+= Diff^2
19                          count++
20                      }
21                  }
```

```
                Add sum to total: total+= sum/count
            }
            Final cost: FP: total/#ControlPoints
        }
    }
}
Free candidate patch GPU memory
```

Listing 6.10: First version of our GPU cost calculation process

The second version (Listing 6.11) increases the block dimension to (4) to allow four threads to run concurrently. This divides up the work so that each thread works on a component of the cost function when looping over the control points. Each of the segments is broken up in two, centred on the control point's location as per Figure 6.2. Because there is now more than one thread doing calculations, we make use of a small amount of shared memory to store the cost value from each threads calculation. A useful observation made is that the only variation in code, where branching occurs, is the calculation of the segment positions. After that the same code path is executed for all four threads concurrently with their results stored in separate shared memory locations. The final step is to synchronise the threads and then have a single thread perform the final summation of the costs from the other ones stored in shared memory. This value is then divided by the number of control points to produce the final cost for this candidate.

```
Inputs: Source Candidates
Output: Calculated costs for candidates – Stored in GPU memory

Allocate memory on GPU {
    Candidate Patch: PatchSize × PatchSize × SizeOf(float)
}
Loop over source patches {
    Loop over number of source transformations {
        Execute candidate patch extraction kernel (Listing 6.9)
        Init kernel: GridDim = (#UserPatches) : GridBlock = (4)
        Execute cost calculation kernel {
            Allocate shared memory: 4 × SizeOf(float)
            Loop over control points {
                Calculate points for segments r and s (Figure 6.2)
                Run profiling for segments r and s {
                    Initialise: sum = 0, count = 0
                    Iterate over all points on segment {
                        Calculate difference: (Diff = Usr_pixel − Src_pixel)
                        Add difference squared: sum+= Diff^2
                        count + +
                    }
                }
                Add sum to total for thread x: total_x+= sum/count
            }
            ~Synchronise threads~
            If thread.id == 0 {
                Total cost: sum(costs in shared memory)
                Final cost: FP: total cost/#ControlPoints
            }
        }
    }
}
```

```
33  Free candidate patch GPU memory
```

Listing 6.11: Second version of our GPU cost calculation process

The third version also makes use of shared memory and the same block dimension of (4), but changes several other aspects. The majority of the algorithm is the same as the second version (Listing 6.11). A total of $(4 \times SizeOf(float))\ bytes$ of shared memory is used for this version. An '*abs*' function was removed because its inverse was placed inside an '*if*' clause which was already executing, to remove the performance hit that an *abs* function has on the GPU pipeline. Another improvement was to not initialise the shared memory to a zero value. Instead, during the looping over the control points, the first iteration will set the value with successive calls adding to the cost. The function that performs the cost calculation with the given edge was unrolled and placed directly into the kernel for a slight performance gain. A single thread is still required to sum the total cost after a synchronisation before the kernel returns.

```
1   Inputs: Source Candidates
2   Output: Calculated costs for candidates – Stored in GPU memory
3
4   Allocate memory on GPU {
5       Candidate Patch: PatchSize × PatchSize × SizeOf(float)
6   }
7   Loop over source patches {
8       Loop over number of source transformations {
9           Execute candidate patch extraction kernel (Listing 6.9)
10          Init kernel: GridDim = (#UserPatches) : GridBlock = (128)
11          Execute cost calculation kernel {
12              Allocate shared memory: 128 × SizeOf(float)
13              Calculate points for segments r and s (Figure 6.2)
14              Initialise: sum = 0, count = 0
15              Iterate over all points on segment {
16                  Calculate difference: (Diff = Usr_pixel − Src_pixel)
17                  Add difference squared: sum+= Diff^2
18                  count + +
19              }
20              Add sum to total for thread x: total_x+= sqrt(sum)/count
21              ~Synchronise threads~
22              If thread.id == 0 {
23                  Total cost: sum(costs in shared memory)
24                  Final cost: FP: total cost/#ControlPoints
25              }
26          }
27      }
28  }
29  Free candidate patch GPU memory
```

Listing 6.12: Fourth version of our GPU cost calculation process

The fourth version (Listing 6.12) attempts to use additional threads, to iterate over the additional control points as separate threads. This means individual threads have no loop structures. We multiply the current block dimension of (4) by a set maximum number of control points of 32 to give a dimension (128). The previous work we were extending had a hard limit of 32 control points, which we adopted as well. However, this hard constraint is not advised for varying patch sizes as the number of control points varies with the size of the patches. We now unroll the loop over all the

control points and use some logic to work out which control point the thread is working on. The shared memory amount is also increased to $(16 \times SizeOf(float))\ bytes$, so each thread has its own piece of memory. The process ends with the single thread totalling all the shared memory blocks. This method proves inefficient as there are not always 32 control points which results in threads being idle and wasting executing time. After the development of some of our advanced GPU implementations we discovered that the system in fact uses no more than three control points for our given patch size. Adjusting values down to four yielded significant improvements in this version and is re-integrated in our advanced implementations in version eight. The explanation for the poor results was the idling of the vast majority of threads, which were not required and a waste of resources.

Advanced GPU versions

The next four GPU versions are different in that only a single loop is used, which iterates over the source patches. We allocate enough memory to hold all the different transformations of a source patch $(10 \times PatchSize^2 \times SizeOf(float))\ bytes$. A kernel with grid dimensions $\left(\frac{PatchSize}{32}, \frac{PatchSize}{32}, 10\right)$ and block dimensions $(32, 32)$ is used to efficiently extract the source patch, perform the required transformations and then store all of them (Listing 6.13). Versions 6-8 share a common grid dimension $(\#UserPatches)$ for the cost kernels. The code now diverges for each of the four advanced GPU versions.

```
1   Inputs: Source File, Patch Offset
2   Output: Candidate patch stored in GPU memory
3
4   Init Kernel: GridDim = (PatchSize/32, PatchSize/32, #Transformations) : BlockDim = (32,32)
5   Execute extraction kernel {
6       Calculate lookup coordinates for sketch for all transformations
7       Calculate memory address for output
8       Extract pixel data from source
9       Store data in user patches memory
10  }
```

Listing 6.13: Overview for the advanced candidate patch extraction kernel

Version five (Listing 6.14) is a modified implementation of version three that makes use of additional blocks, which runs each of the ten transformations. The grid dimensions are expanded to $(\#UserPatches,\ 10)$ with the block dimensions still set at four threads. There is a slight speedup due to splitting up the workload to a greater degree.

```
1   Inputs: Source Candidates
2   Output: Calculated costs for candidates – Stored in GPU memory
3
4   Allocate memory on GPU {
5       Candidate Patch: PatchSize × PatchSize × #Transformations × SizeOf(float)
6   }
7   Loop over source patches {
8       Execute candidate patch extraction kernel (Listing 6.13)
9       Init kernel: GridDim = (#UserPatches, #Transformations) : GridBlock = (4)
10      Execute cost calculation kernel {
11          Allocate shared memory: 4 × SizeOf(float)
12          Loop over control points {
13              Calculate points for segments r and s (Figure 6.2)
```

```
            Initialise: sum = 0, count = 0
            Iterate over all points on segment {
                Calculate difference: (Diff = Usr_pixel − Src_pixel)
                Add difference squared: sum+= Diff^2
                count + +
            }
            Add sum to total for thread x: total_x+= sqrt(sum)/count
        }
        ~Synchronise threads~
        If thread.id == 0 {
            Total cost: sum(costs in shared memory)
            Final cost: FP: total cost/#ControlPoints
        }
    }
}
Free candidate patch GPU memory
```

Listing 6.14: Fifth version of our GPU cost calculation process

Version six (Listing 6.15) is an extension to version five, but instead of using more blocks for processing we exploit part of the initialisation in the feature profiling algorithm. For each of the control points in the user patch, the four threads work out their corresponding offset location for each of the segments. Then they compare the height profiles with the candidate patch. Each of the threads will calculate the initial part and then when it comes to comparisons with the candidate patches, we add an additional loop which will check each of the candidates. The shared memory is increased so that each thread can store its information for each of the ten candidates it is evaluating. The process ends with a single thread totalling the costs from each of the threads for each of the candidates, producing ten cost values. Due to the additional looping done by single threads, this version works best when there are a large number of user patches to evaluate to ensure the GPU is fully saturated to offset the looping overhead.

```
Inputs: Source Candidates
Output: Calculated costs for candidates – Stored in GPU memory

Allocate memory on GPU {
    Candidate Patch: PatchSize × PatchSize × #Transformations × SizeOf(float)
}
Loop over source patches {
    Execute candidate patch extraction kernel (Listing 6.13)
    Init kernel: GridDim = (#UserPatches) : GridBlock = (4)
    Execute cost calculation kernel {
        Allocate shared memory: 4 × 10 × SizeOf(float)
        Loop over control points {
            Calculate points for segments r and s (Figure 6.2)
            Initialise: sum = 0, count = 0
            Iterate over all points on segment {
                Calculate difference: (Diff = Usr_pixel − Src_pixel)
                Add difference squared: sum+= Diff^2
                count + +
            }
            Add sum to total for thread x: total_x+= sqrt(sum)/count
        }
        ~Synchronise threads~
        If thread.id == 0 {
```

```
            Total cost: sum(costs in shared memory)
            Final cost: FP: total cost/#ControlPoints
        }
    }
}
Free candidate patch GPU memory
```

Listing 6.15: Sixth version of our GPU cost calculation process

Version seven (Listing 6.16) combines features from versions five and six, taking the additional looping from six and using the extra dimensions from five for the processing. The grid dimensions are still set to the number of user patches but the block dimensions are extended to $(4,10)$. The shared memory is thus increased to allow for the ten transformations for each of the four components. But instead of adding an extra loop as done in version six, we use the extra dimension of threads to calculate the costs. While this version performs better than version five, it can at best perform the same or slightly worse than version six. It also requires terrains with large numbers of user patches to fully saturate the GPU with enough independent blocks.

```
Inputs: Source Candidates
Output: Calculated costs for candidates – Stored in GPU memory

Allocate memory on GPU {
    Candidate Patch: PatchSize × PatchSize × #Transformations × SizeOf(float)
}
Loop over source patches {
    Execute candidate patch extraction kernel (Listing 6.13)
    Init kernel: GridDim = (#UserPatches) : GridBlock = (4, #Transformations)
    Execute cost calculation kernel {
        Allocate shared memory: 4 × 10 × SizeOf(float)
        Loop over control points {
            Calculate points for segments r and s (Figure 6.2)
            Initialise: sum = 0, count = 0
            Iterate over all points on segment {
                Calculate difference: (Diff = Usr_pixel − Src_pixel)
                Add difference squared: sum+= Diff^2
                count + +
            }
            Add sum to total for thread x: total_x+= √sum/count
        }
        ~Synchronise threads~
        If thread.id == 0 {
            Total cost: sum(costs in shared memory)
            Final cost: FP: total cost/#ControlPoints
        }
    }
}
Free candidate patch GPU memory
```

Listing 6.16: Seventh version of our GPU cost calculation process

Our final GPU version (Listing 6.17) takes another look at the less impressive version four and attempts to improve its performance. We optimised the looping mechanics to prevent unrequired threads from executing code thus resulting in it performing better than the original version four. We also performed testing to determine a better limit for the control points and never observed more

than four control points for any of the input files we made use of. The improvements from version seven are also incorporated, which brings the block dimensions to $(16,10)$. This improves substantially on version four and doubles the speed of version seven. We then investigated making use of texture memory to improve performance further.

```
1  Inputs: Source Candidates
2  Output: Calculated costs for candidates – Stored in GPU memory
3
4  Allocate memory on GPU {
5      Candidate Patch: PatchSize × PatchSize × #Transformations × SizeOf(float)
6  }
7  Loop over source patches {
8      Execute candidate patch extraction kernel (Listing 6.13)
9      Init kernel: GridDim = (#UserPatches) : GridBlock = (32, #Transformations)
10     Execute cost calculation kernel {
11         Allocate shared memory: 32 × 10 × SizeOf(float)
12         Calculate points for segments r and s (Figure 6.2)
13         Initialise: sum = 0, count = 0
14         Iterate over all points on segment {
15             Calculate difference: (Diff = Usr_pixel − Src_pixel)
16             Add difference squared: sum+= Diff^2
17             count + +
18         }
19         Add sum to total for thread x: total_x+= sqrt(sum)/count
20         ~Synchronise threads~
21         If thread.id == 0 {
22             Total cost: sum(costs in shared memory)
23             Final cost: FP: total cost/#ControlPoints
24         }
25     }
26 }
27 Free candidate patch GPU memory
```

Listing 6.17: Eighth and final version of our GPU cost calculation process

GPU Texture Memory

We added an option to enable the use of texture memory on the GPU for the user sketch and source images. This affects the user patch extraction kernel, along with the candidate extraction kernel used in each of the GPU versions for cost calculation. Section 8.1.4 presents the results obtained from using texture memory. The implementation requires few code changes from the implementations in Listing 6.9 and Listing 6.13, a bound texture reference is needed for the lookups instead of passing in an array. The system is capable of interchangeably using texture memory or the global memory array directly during synthesis.

6.4.4 Storing Best Candidates

Once the cost kernels have been executed for all of the source files, the data needs to be sorted and only the best set of candidates retained. This set then needs to be relayed to the CPU for the merging process to begin. We develop two different implementations for this process; the first simply transfers all the raw cost data to the CPU for sorting, while the second performs the sorting on the GPU.

The first implementation relies entirely on the CPU to perform the sorting operation. This requires the raw cost data be transferred back to host memory. We made use of the built in C++ stable sort algorithm. Once transferred, we create a data-structure that associates the candidate indices and cost values. This is now sorted in ascending order of cost, with the five lowest cost candidates being saved. This processed is repeated for all of the user patches and the results sent for merging. Transferring the raw data to the host has a high time cost due to the large amount of data being transferred, sorting on the GPU and transferring only the best-set data would, in theory, be more efficient.

```
1  Inputs: Candidate costs for each user patch
2  Output: Set of 5 best candidates for each user patch
3
4  Allocate memory on GPU (#UserPatches × 5 × SizeOf(int))
5  Init kernel: GridDim = (#UserPatches) : BlockDim = (1)
6  Execute sort kernel {
7      Loop over number to keep: 5 {
8          Loop over #CandidateCosts {
9              Store lowest cost candidate index
10             Set cost to infinity to indicate stored
11         }
12     }
13 }
14 Create array on host (#UserPatches × 5)
15 Copy data from device to host
16 Build best set from index values for #UserPatches
```

Listing 6.18: Algorithm for sorting candidates based on cost in ascending order

The GPU sorting implementation has two different modes: the first is our own written kernel (Listing 6.18) to sort the data and the second uses an external library, Thrust (2013), which includes functions optimised for sorting. Our sorting kernel has a grid dimension $(\#UserPatches)$ and block dimensions (1); this provides a single thread that runs over two loops. It is based on the observation that a full sort is not required. The outer loop iterates over the number of candidates we chose to retain (5), searching the entire set of candidates looking for the lowest cost. When found the index for the candidate is recorded and its cost set to infinity before the next iteration. Once the kernel is finished executing the results are transferred to the host, so the CPU can assemble the best set candidates and pass them to the merging process. The use of the Thrust (2013) library provides a function that is highly optimised, to replace our sorting kernel, while the rest of the process remains the same. The results for all three variants of this process are presented in section 8.1.5.

6.4.5 Merging

Now that the best-set of candidates is available back on the CPU side, they can be processed for merging. The merging process is not ported to the GPU in our research and is thus done on the CPU. Section 6.5 describes the process for merging; while the CPU is processing some of the data for merging, the GPU continues building up new sets of data asynchronously as part of the blocked design. This splits the workload into smaller chunks so that the CPU can be processing a chunk while the GPU is preparing the next; this is described in section 6.6.

6.5 Feature Merging

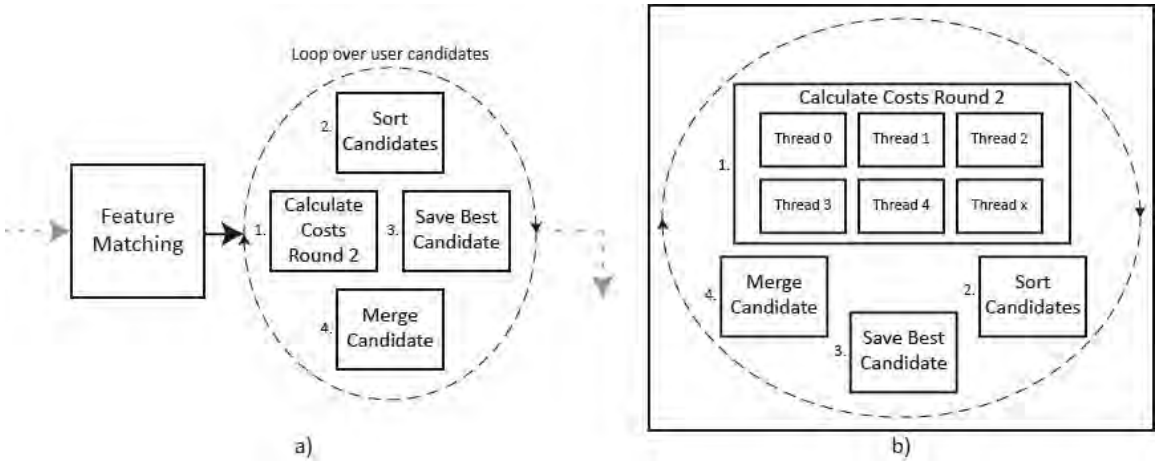


Figure 6.6: Overview of the feature merging pipeline: a) Single-threaded pipeline, b) Internal block for multithreaded version.

The merging process used in our system remains largely unchanged from that of Tasse et al. (2011), as described in section 4.3. Figure 6.6 shows an overview of the merging process with the parallelisable components separated out. Once the matching system has completed for all the source files and the best set is produced, the merging process takes over and a second more comprehensive round of cost calculations is undertaken. This is part of the merging core because the information already synthesised in the output terrain is used. The combination of SSD and Noise Variance allows a recalculation of the cost values based on the synthesised data. This list is then resorted and only a small subset of five candidates is evaluated against the Graph-cut cost due to the large computational overhead it carries. The overall best candidate is selected and passed onto the merging process.

The merging process is responsible for placing the best candidate into the output terrain. Simply pasting the candidate directly into the output terrain will result in obvious artefacts with a very pronounced straight boundary due to the difference in pixel values. For the result to appear seamless, a combination of techniques is used. First a Graph-cut (Section 4.3.1) is performed on the overlapping area to find an optimal cut path. However, this seam remains visible, which requires further processing. Shepard Interpolation (Section 4.3.2) is used to deform the pixel data around the seam so that the pixels on both sides have similar values. This process works well with a top-down view of the 2D image showing no visible artefacts. However, a 3D rendering of the terrain shows that the gradient values along the seam are not matched, revealing an unnatural discontinuity. Tasse et al. (2011) correct this artefact by performing seam removal on the image gradient field of the overlapping region. The final elevations for this patch are reconstructed from the modified gradient field by solving a Poisson equation (Section 4.3.3). The patch has now been successfully placed into the terrain free from any visual artefacts and the process can proceed to the next patch.

The cost computation part of this process is enhanced with multithreading to better distribute the workload. The recalculation of the costs for all the best-set candidates is divided equally amongst a set of threads, after which a single thread sorts the data. The Graph-cut cost is only executed on the top five candidates, each of which is run in a separate thread. The actual merging of the patch into the output terrain is done on a single thread.

6.6 Optimisations

Throughout the implementation of the feature synthesis a number of optimisations were applied to improve the system. The two cost calculation rounds are one such example; these were described in sections 6.2 and 6.3. Due to the system architecture changes required for multiple input sources it is not possible to run some of the cost functions without data from the output terrain.

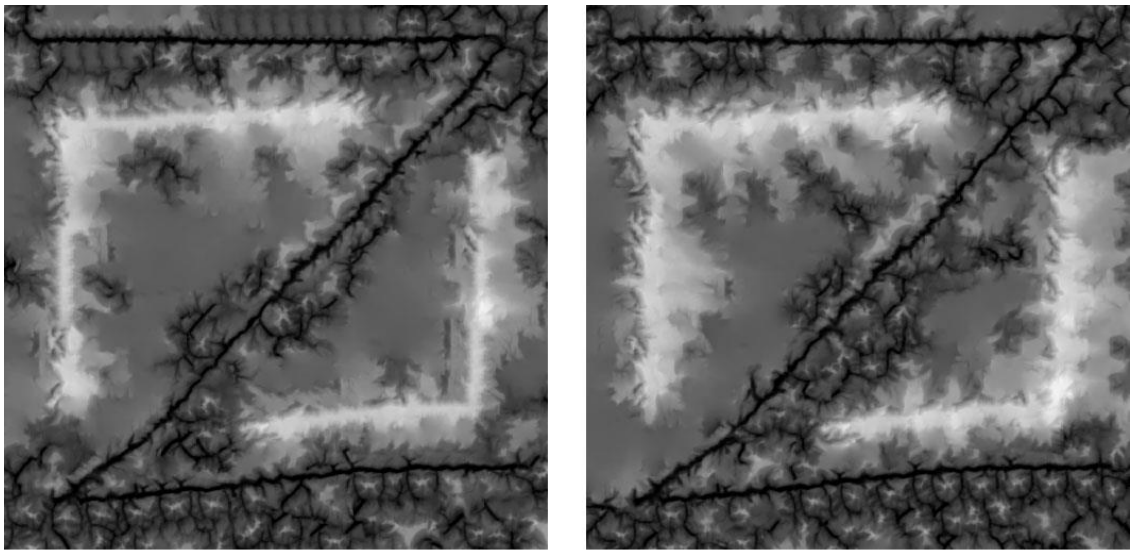


Figure 6.7: Example of repetition in output terrain. (a) Repetition with adjacent patches (b) Repetition check implemented to overcome this issue

There is a chance that during synthesis the same candidate may be chosen for successive user patches. This leads to a repetition of adjacent patches in the output terrain, which is visually jarring. In order to fix this issue we keep track of the last placed candidate, which is penalised with a very high cost to prevent it being selected, unless there are no other matches. Since our system only retains a maximum of five candidates for each user patch, we have a limited amount of data to choose from when preventing repetition. This is a trade-off with the amount of memory available to the system as keeping more candidates would require much more memory and increase the time to complete synthesis. Figure 6.7 shows an example of such repetition artefacts as well as the same user sketch being synthesised with our fix to prevent repetition in adjacent patches. This improvement only retains a reference to the last placed patch, which does not help if adjacent patches are not processed successively. As such the repetition artefact may still occur. To address this, a secondary data-structure could be maintained and queried at merge time to ensure that the same patch was not being placed adjacent or even in close proximity. This would add further complexity for the system and was chosen to be left for future work. Implementing this concept would require us to store more candidates as it would shift the limitation of repetitiveness to how many patches there are to choose between.

During the feature extraction process, the distance between successive nodes may be as little as a few pixels, which will result in a significant overlap of patches. This would be extremely wasteful since we would end up synthesising the same area multiple times. To address this we only generate patches that have a minimum linear distance of $^1/_{16}\,th$ the patch size between neighbouring nodes. This can dramatically reduce the number of user patches, especially in larger terrains where the

feature extraction process produces errors with edges forming closely spaced parallel lines (Figure 6.8). Improvements to the extraction algorithm could potentially solve this issue.

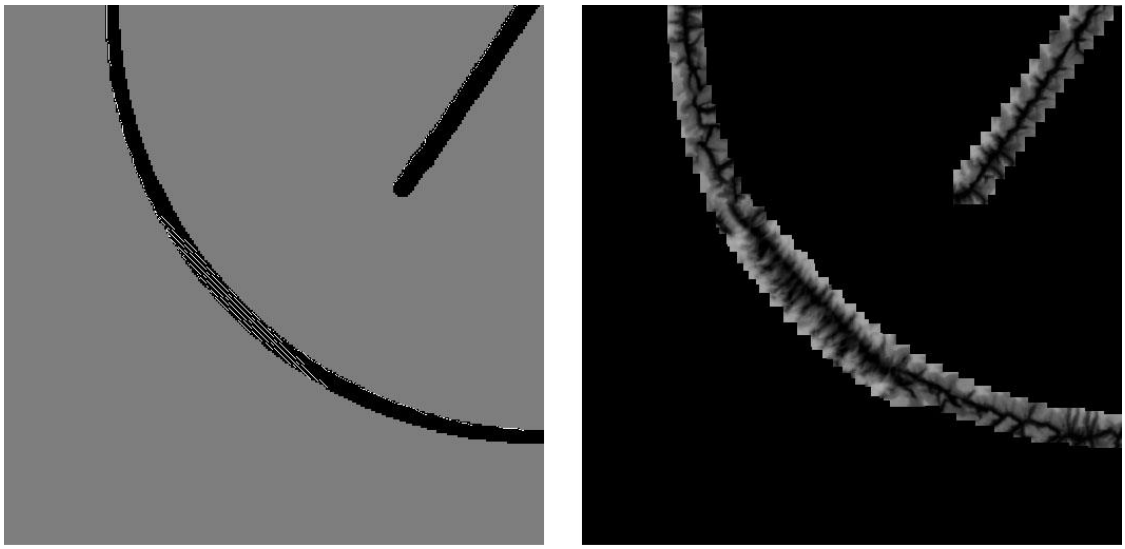


Figure 6.8: (a) Example of error with feature detection engine forming multiple parallel lines. (b) This results in heavy overlaying of patches, which wastes performance.

Pre-loading all the data into memory means the system only needs to read it from disk once, which saves on the transfer time from disk. This allows us to improve performance over the system designed by Tasse et al. (2011). Their system extracts the pixel data for every candidate from the source file and stores it in memory during the cost calculations. This method quickly exhausts all available system memory and could potentially crash when processing source files with large numbers of patches. A source file with 5,000 patches would expand to 50,000 candidates under the different transformations, which would require 800MB worth of memory to store just the 64×64 pixel patch data. There is also a large time overhead in extracting all the candidates before computing their cost. We developed our system to keep only the original source file in memory and do direct memory lookups during cost calculations. This allows us to process very large source files with thousands of candidates. Once the best patch is located by the system, its pixel data is extracted and passed to the merging subsystem.

When developing the GPU implementation many optimisations from the CPU implementation were retained. Pre-loading the data onto the GPU device allows for much faster synthesis as the largest overhead with GPU programming is copying data between the host and device. We have also optimised the GPU kernels through proper use of the available memory types (Section 6.4.3).

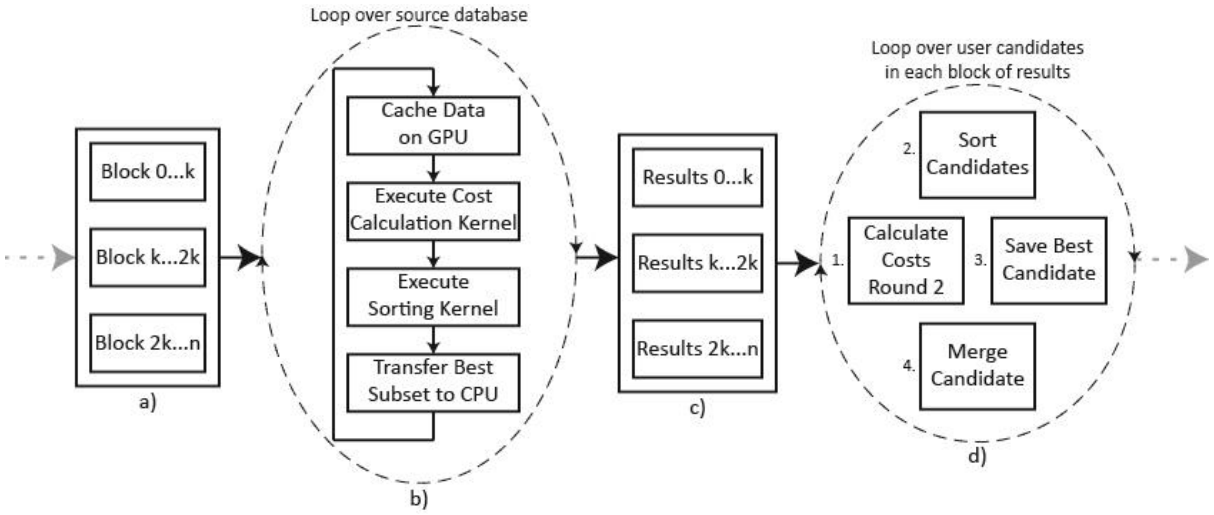


Figure 6.9: Illustration of blocked design for candidate processing. a) A queue of blocks of length k that are sequentially processed by the algorithm in b) on the GPU. Results form a queue c) which is processed by the CPU in d)

The last important optimisation is the development of a blocked design, which splits the work into a series of chunks for processing the cost calculations (Figure 6.9). This design became necessary for very large user sketches with hundreds or even thousands of user patches. Our algorithm is designed to process all the user patches in order to store the best set of candidates before the merging process. But processing all the data in the limited high-speed memory available on a GPU is not possible. Thus, it becomes necessary to divide the task into a series of smaller chunks. Another benefit of this blocked design is that after a block has been processed by the GPU, its results are transferred back to the CPU to begin the merging process. At that point the next block is processed by the GPU. This allows for a balance of the CPU and GPU processing capabilities. There is a small overhead with managing the various processes but as shown in section 8.1.5 there is a reduction in the synthesis time when balancing the tasks asynchronously. As the CPU is slower at processing than the GPU, these changes ensure that the CPU is never idle during the synthesis process.

There are several improvements that have not yet been implemented and are listed in section 9.2 for the future work. Further GPU optimisations can be achieved by exploiting the functionality provided on new generation devices. Next we look into the non-feature synthesis process.

7 Non-Feature Synthesis

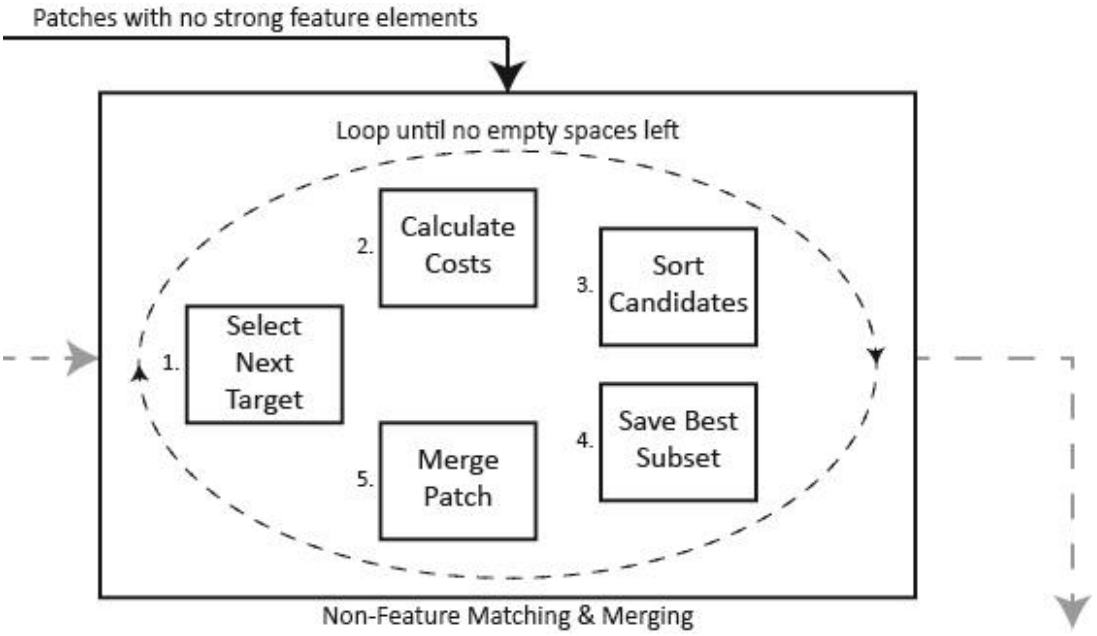


Figure 7.1: Non-feature synthesis pipeline showing flow of data for the Non-Feature Matching & Merging block of our system (Figure 5.1)

After all the user features have been matched by the system there are areas in the output that have no data and appear as *holes* in the terrain. Non-feature synthesis is the process of filling these holes with data from the source files that contain no strong features. This process sequentially selects an area, matches a suitable candidate and then merges the data. This is sequential due to the matching process relying on data in the output terrain, which is updated with each iteration. There are few subroutines that can be accelerated through parallel processing, such as the cost calculations, which we implemented on the GPU. Figure 7.1 shows an overview of the non-feature synthesis process, which is described in the rest of this chapter.

7.1 Candidate Extraction

Candidate patches that contain no strong feature data are required for the non-feature synthesis. These patches are extracted from the source files using a modified feature extraction algorithm. As we need the overall synthesis results to be repeatable, the extraction algorithm cannot have any randomness in the selection process, unless a seeded random number is used for repeatable results. We designed our system to examine fixed intervals along the x and y axes of the source files for suitable candidates. This gives us a fixed number of candidates and the interval is calculated as follows,

$$Interval = (SrcWidth - (2 \times PatchSize^2) \div 8)$$

where $SrcWidth$ is the width of the source file. This is calculated for both axes, which gives us the coordinates to iterate over. These form the centre points for the patches to be extracted. Each of these locations is then compared to the feature extraction node list to make sure that the patch location is not within $\frac{PatchSize}{8}$ pixels of a feature node. This ensures that we avoid patches with

strong features being used. Finally, each of the valid patches are extracted from the source file and then the next source file is then examined. We purposely chose a small number of candidates per source file to limit the complexity of the overall system and the impact on performance. This was due to the non-feature synthesis process not being ported to the GPU due to time constraints; this is noted as future work in section 9.2.

Candidate extraction is run as a pre-processing step at the beginning of non-feature synthesis; all of the valid patches from all source files are cached in memory for later processing in this stage. To increase the number of candidates for the system, each patch is rotated and mirrored in the x and y planes similarly to feature synthesis. These candidates are now examined during the matching stage.

7.2 Candidate Matching and Merging

The system now loops continuously until there are no longer any holes in the output terrain. There is no way of pre-computing the number of non-feature patches that need to be placed to complete the process. This is because the boundary is continually changing with each merge operation, with each placing a different amount of data into the output. Patches are placed at locations on the boundary only so as to provide some valid data to facilitate merging. This unpredictability prevents us from determining the runtime. A single iteration comprises selection, matching and merging processes. The system has no smallest size of hole that it will match and will handle a hole consisting of a single pixel, which ensures the output contains no invalid data. Once all the holes have been filled the synthesis process concludes and the final image is displayed.

7.2.1 Selecting Target Patch

For the first iteration the system needs to evaluate the output terrain to identify all the locations that are on the boundary of placed data and an empty area. This is done once over the whole image and then, after each merging operation, only the affected area is updated. Each of these boundary locations has an associated priority value, which is based on the already placed data around the patch location. At the start of non-feature synthesis, the current output image is processed to build up a list of priority values for the boundary pixels (Listing 7.1). This list is sorted at the start of each iteration, the location with the highest priority is selected for the current matching and merging operation. The details of this algorithm are discussed in section 4.2.2.

```
1  Loop over output pixels {
2      if pixel valid && on boundary {
3          Calculate priority and add to list
4      }
5  }
```

Listing 7.1: Algorithm overview for building boundary dataset

7.2.2 Matching – Cost Functions

Non-feature synthesis makes use of three of the same cost functions used for feature synthesis, namely sum-of-squared differences, noise variance and graph-cut. These cost functions are well suited as they make use of already placed data in the output terrain. We again make use of two rounds of cost calculations. This lets the system only compare a subset of candidates from the first round against the computationally expensive second round. The first round compares the candidate patches with the Sum-of-Squared Differences (SSD) (Section 6.2.2) and Noise Variance (Section 6.2.3) cost functions. After the candidates are evaluated and sorted, the five lowest cost candidates

are evaluated against the Graph-cut cost (Section 6.2.4) function and the best patch is selected for merging.

7.2.3 Matching – CPU Implementation

Our CPU implementation for non-feature synthesis is adapted from Tasse et al. (2011), with additional enhancements. An overview of this process is provided in Listing 7.2 and the CPU pipeline is illustrated in Figure 7.1. The process starts by extracting the candidate patches from all the source files and storing them in memory (Section 7.1). The boundary dataset is now initialised. This evaluates all the points that lie on the boundary of synthesised data and a hole and computes an associated priority value (Listing 7.1). The process now enters a hole-filling loop, which only concludes when there are no remaining holes in the output terrain.

```
1  Extract candidates from all source files
2  Populate boundary dataset
3  Loop until no holes {
4      Select target patch
5      First round cost calculation
6      Sort results
7      Second round cost calculation
8      Sort candidates
9      Merge best patch
10     Update boundary dataset
11 }
```

Listing 7.2: Algorithm overview for the CPU non-feature matching implementation

The first step is to select a target patch for matching (Section 7.2.1). Once a patch is found, its pixel data, where it exists, is extracted from the output terrain and passed to the first round of cost calculations. For each candidate patch, the SSD is calculated for the areas that correspond to valid data in the target patch for each of the different transformations. The noise variance for all the candidate transformations are identical, thus only one comparison is required and the result is added to each of the candidates total cost. After all the candidates have been evaluated, they are sorted in ascending order and the five with the lowest cost are run through a second round of more comprehensive cost calculations. The second round uses the Graph-cut cost function to measure the quality of the merge for the patch. The results are sorted once again with the overall cheapest patch being selected for actual merging. The boundary dataset is now updated around the modified location before the process repeats, continuing until all the holes are filled.

The only parts of this implementation that benefit from parallelism are the cost calculations, which could be distributed amongst a group of threads. However, due to time constraints we chose not to implement a parallel CPU implementation and instead invested time in a basic GPU implementation where we parallelise the cost calculation stage. There is room for improvement of non-feature synthesis through better optimisation and parallelism, but this is left for future work (Section 9.2).

7.2.4 Matching – GPU Implementation

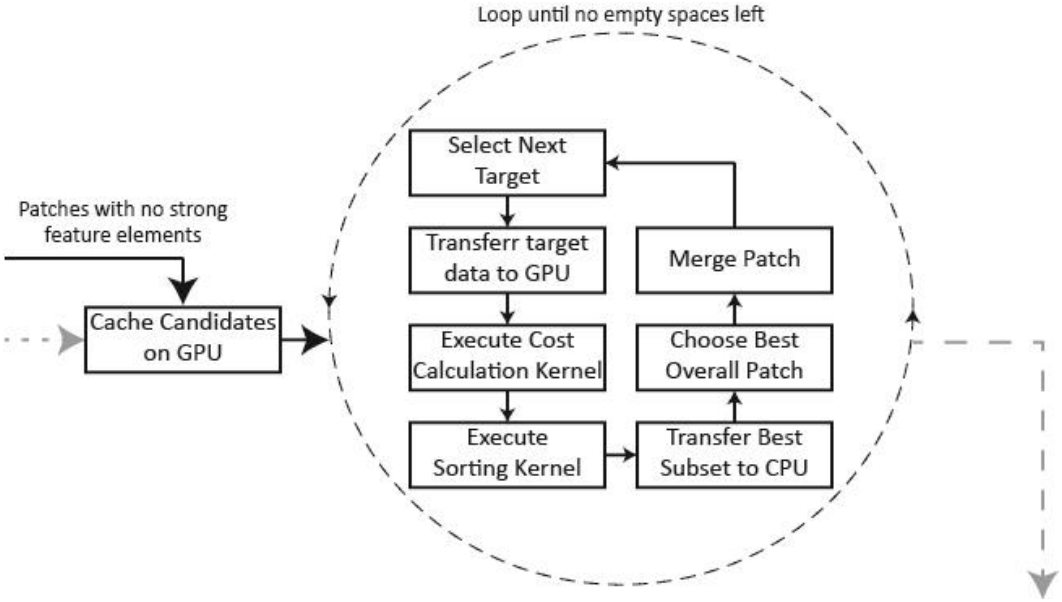


Figure 7.2: Overview of the GPU non-feature matching pipeline. Candidates are cached on the GPU initially. The system then loops until all 'holes' are filled. GPU acceleration is used to calculate the costs with the rest being done on the CPU.

The GPU implementation we have developed is a basic attempt at parallelising non-feature synthesis; only the cost calculation and sorting are performed on the GPU with the rest done on the CPU. An overview of this process is presented in Listing 7.3. The process starts with caching all of the source files on the GPU, but this is normally skipped as the source files will have already been cached on the GPU as part of feature synthesis. The candidate extraction process starts with allocating memory to store all the candidates in GPU memory. We then loop over all the source files and calculate all the valid candidate positions as per section 7.1. These positions are now transferred to the GPU and an extraction kernel is executed, with grid dimensions of $\left(\frac{PatchSize}{32}, \frac{PatchSize}{32}, \#ValidCandidates\right)$ and block dimensions $(32, 32)$. The pixel data is extracted from the source file and stored in GPU memory before the next source file is processed.

```
1   Cache source data on GPU (CPU -> GPU)
2   Execute candidate generation (GPU)
3   Populate boundary dataset (CPU)
4   Loop until no holes {
5       Select target patch (CPU)
6       Transfer target patch data (CPU -> GPU)
7       First round cost calculation (GPU)
8       Sort results (GPU)
9       Transfer best-set to CPU (GPU -> CPU)
10      Second round cost calculation (CPU)
11      Sort candidates (CPU)
12      Merge best patch (CPU)
13      Update boundary dataset (CPU)
14  }
```

Listing 7.3: Algorithm overview for the CPU non-feature matching implementation

The boundary data set is now populated, as in the CPU implementation, and this concludes the pre-processing stage. The system will loop until all the holes are filled and synthesis is complete. The CPU selects the next target patch based on the priority values for the boundary pixels. The

coordinates are sent to the GPU and used to extract the target patch pixel data from the output terrain and transfer it to GPU memory. Due to the output changing after each merge operation, which is done on the CPU, there is no copy of it stored on the GPU. A CUDA kernel to compare the costs is now executed in two parts, the first part calculates the SSD and the second the noise variance of the target patch. The SSD calculation has grid dimensions of $(\#ValidCandidates, \#Transformations)$ and block dimensions $(PatchSize)$, which allows each of the candidates and its associated transformations to be run as separate blocks. With the number of threads equalling the patch dimensions, each one has an assigned x coordinate and calculates the SSD value for all of the y coordinates. There is now a synchronisation phase, because the threads all execute independently and we require all threads to have finished executing and writing their results to memory. A single thread now sums up the SSD values from each of the other threads and stores the final SSD cost in an array on the GPU.

The noise variance is then calculated for all the candidates using a kernel with grid dimensions $(\#ValidCandidates)$ and block dimensions (1). Because the noise variance for a candidate is the same irrespective of its transformation, we need not calculate it for each transformation. After the cost is calculated the total is added to the cost array from the SSD stage, the cost is also added to each of the candidate transformation cost values. This process completes when all the source files have been evaluated. The final cost array is now sorted on the GPU, in ascending order. We designed a simple kernel for this process but also developed a version that makes use of the Thrust (2013) library for improved performance. This is similar to feature synthesis candidate sorting (Section 6.4.4). The best subset of five candidates is now returned to the CPU where they undergo the second round of cost calculations. This was done on the CPU, as our simple port of the code to GPU resulted in a negative speedup due to the algorithm complexity.

The graph-cut cost function is now used to compare each of these five candidates, which are then sorted with the lowest cost one being chosen as the best overall. This patch is sent to the merging process to be placed into the output terrain. When this process is run for the first time candidate generation process is required, but on successive iterations this step is skipped and only the calculation process is run with the new target patch. Once all the holes are filled the data is deleted from the GPU memory.

7.2.5 Merging

Once the best patch has been selected it is merged into the output terrain. This process is almost the same as that detailed in feature synthesis (Section 6.5). The difference is that the merge step does not conclude the synthesis. Since the merging process alters the output terrain, the boundary needs to be updated for target selection. This is done using the location of the patch and the process then continues with selecting a new target. If there are no remaining targets the synthesis concludes returning the final result to the user.

7.3 Optimisations

During the implementation of non-feature synthesis a number of optimisations were applied to improve the synthesis process. An important optimisation improved the memory utilisation of candidate extraction: in order to limit the amount of memory required, only a single patch is stored for each candidate. Thus, only 90° rotations and mirrors can be supported since 45° rotations require data from the surrounding patch. This reduces the number of candidates per patch to six, down from

ten for feature extraction (Section 6.1). This is more efficient as we can easily change the direction when reading the pixel data and we do not need to perform interpolation or store a larger area to include pixels on the edges of a 45° rotation.

By pre-processing all the source files and keeping the candidates cached in memory, we are able to quickly evaluate all of them against the selected target patch without the need to loop over the source files. This is possible due to the low number of candidate patches taken from each source file, which keeps the memory requirements relatively low. This is important for the GPU implementation as swapping data in and out would negate any performance gains. We kept the number of candidates low since the process was not ported to the GPU and increasing the candidates would hamper performance more. As part of the future work, the number of candidates could be increased after the system is ported to the GPU.

As part of the optimisation process for feature synthesis, we developed a multithreaded graph-cut cost function. As the data input for this function is the same for the non-feature synthesis, we are able to reuse this to give a slight performance increase.

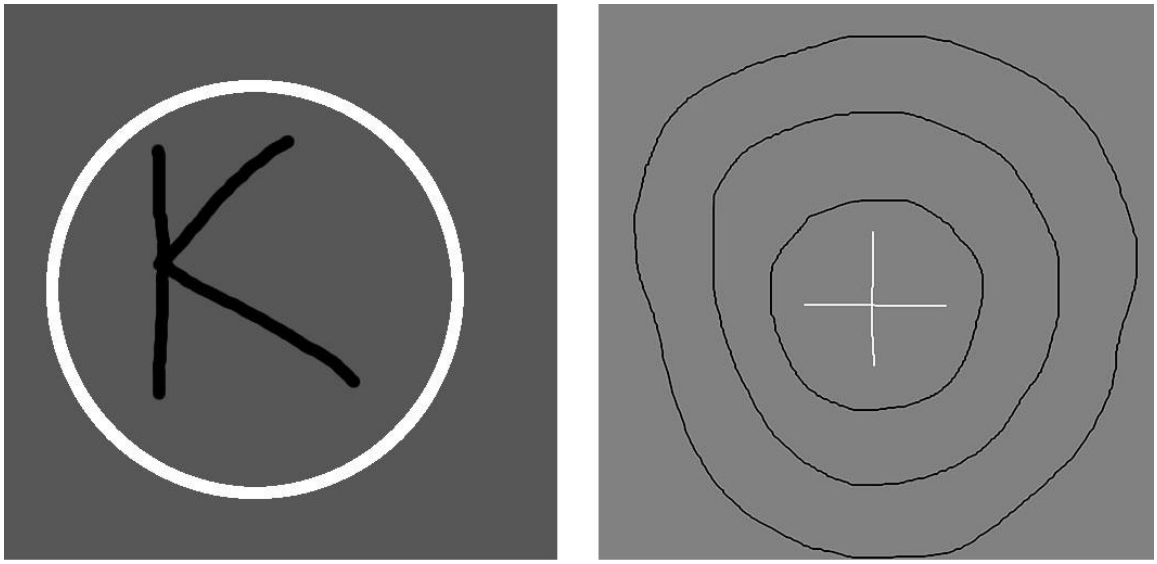
8 Results

In this chapter we present the results and evaluate the extent to which we met our initial goals. Our research has two primary goals; firstly to extend the work of Tasse et al. (2011) by utilising a large collection of source files for the synthesis operation; and secondly, to improve system performance to counteract the inclusion of many more source files and allow the creation of larger more complex terrains. Evaluation takes the form of *visual assessment* of the terrains generated by our system, compared to the previous system as well as the use of *performance metrics* to evaluate the speed.

The test system we utilised features an Intel Core i7 processor with 16GB of RAM and an NVIDIA GeForce GTX 660 Ti GPU. All experiments were run on a fresh reboot of the computer with minimal other processes running; tests were run ten times with the average time reported to allow for representative performance figures. We designed two test images for synthesis, a small 512×512 one and a large 5000×5000 one (Figure 8.1). These have a different number of user candidates to match against to increase the complexity (Table 8.1) as calculated from the feature extraction algorithm. We have separated the ridges and valley runtimes to show values for different numbers of features. If the number or ridges and valleys were the same then near identical runtimes would be observed as the algorithm is the same.

	Small Terrain	Large Terrain
Dimensions	512×512	5000×5000
Ridges	38	68
Valleys	18	900

Table 8.1: Number of detected user features patches and dimensions of the two main test terrains we use. Difference is ridge/valley count is determined by feature extraction and dependant on sketch used.



a) b)

Figure 8.1: The two test images used for evaluation. a) The small 512×512 terrain. b) The large 5000×5000 terrain. The white lines represent ridges with the black lines being valleys as detected by the system.

The results are organised into three sections. The first two sections cover each of the main components of the synthesis engine, namely feature and non-feature synthesis. The third section covers the system as a whole, where the comparison with the previous work is discussed. All the stacked column charts in this section provide the runtime in *seconds* with the columns made up from the major contributing stages of the algorithm; *Patch Matching* and *Patch Merging*. Speedup graphs are also presented to show the performance gains obtained.

8.1 Feature Synthesis

We began by evaluating the feature synthesis component of our system. As feature synthesis is the first part of system, the images we generate contain *holes,* which are filled in the next stage of synthesis. We start off with a very basic CPU implementation, which is based directly off Tasse et al. (2011). Each GPU test builds upon the previous one with, the first version being unoptimised and each of the subsequent optimisations being covered in their own subsection. Some of the optimisations make use of different tests; such differences are noted in the description.

8.1.1 Sequential CPU versions

The first test we conduct compares our two sequential CPU versions (CPU v1 and CPU v2) to evaluate which one is better suited for further development. The first version closely matches that of Tasse et al. (2011), which results in repeated loading and unloading of the source files into memory. Our improved version inverts the loop processing logic, thus only loading each source file once, as well as caching the user patches in memory for faster access.

From the results in Figure 8.2 we observed that there is very little speedup overall for our second version. If we look at the raw runtime values in Table 10.1 we see a dramatic reduction in time spent regenerating the source candidates, especially on the larger terrain. This value, however, is very small and insignificant in the overall synthesis. Since the source files are only loaded once, the time for pre-processing is the same across different terrain complexities. This result motivated us to continue the development of the second version, as it is expected to scale to both larger terrain sizes and a larger number of source files. There is only one disadvantage to the architecture change between the two versions. The first one evaluates every user patch against all source candidates before selecting the best one and merging it. This allows for already merged data to be taken into account to improve selection criteria and result in a better overall merged patch. In order to compensate for this, we keep a list of possible matches in our second version. This list of candidates is evaluated after the matching process completes in which merging is conducted and the candidates are re-evaluated based on information from placing previous patches. Through testing we found very little visual difference between our two versions.

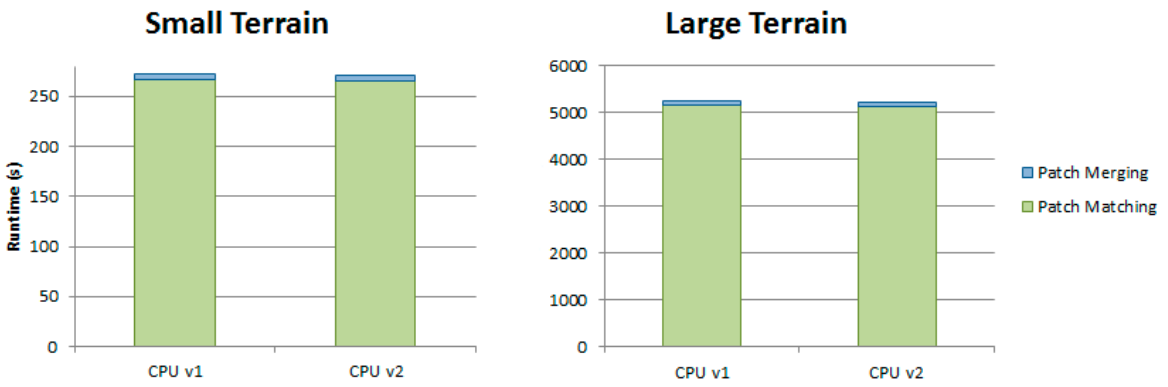


Figure 8.2: Runtime chart comparing the two main CPU implementations. These two implementations have very similar runtimes despite the large architectural changes between them. Table 10.1 gives the runtime numbers in a table and reveals that CPU v2 is slightly faster than v1.

8.1.2 Single versus Multi-Threaded CPU

Based on the results of our sequential implementations we choose to further develop our second version and utilise additional threads to reduce the synthesis time. The whole system could not be multithreaded, or more specifically the iterative merging process, where each successive patch is calculated based on information from already placed patches and thus introduces and order dependency. Our test system features four cores with Intel's Hyperthreading giving a total of eight threads that can be run concurrently. The area targeted for multithreading was the cost calculation process which accounts for over 95% of the total runtime; thus any improvement would have a significant impact. In order to accommodate multiple threads, minor changes were required to control the distribution of the workload. From the results in Figure 8.4, we observed a 1.7 times speed improvement, which translates to a 35 minute reduction in time on the large terrain.

Our multithreaded implementation was relatively basic and unoptimised, but it served as a proof of concept for our GPU implementations. From the results it is evident that there was much potential for improving the running time by parallelising the cost calculation process

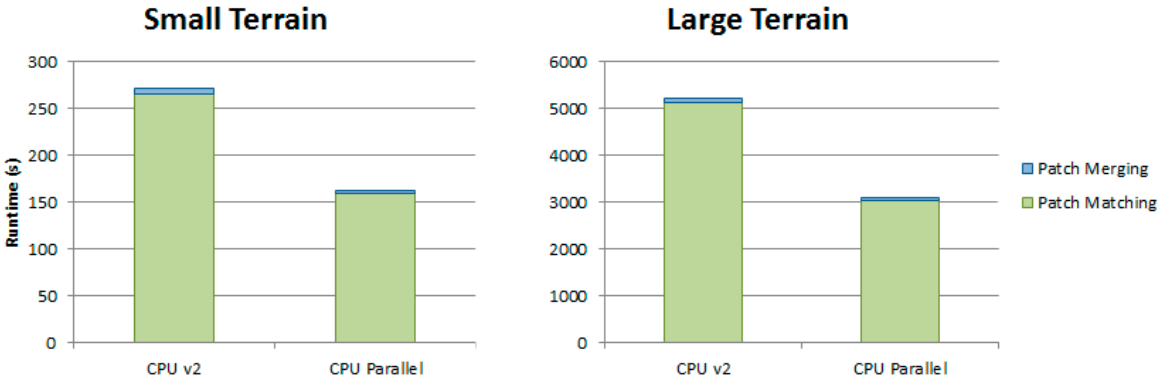


Figure 8.3: Runtime results comparing the parallel CPU implementation against CPU v2. Here we observe a large reduction in synthesis time almost reducing it by half on the large terrain. Full runtime values are presented in Table 10.2.

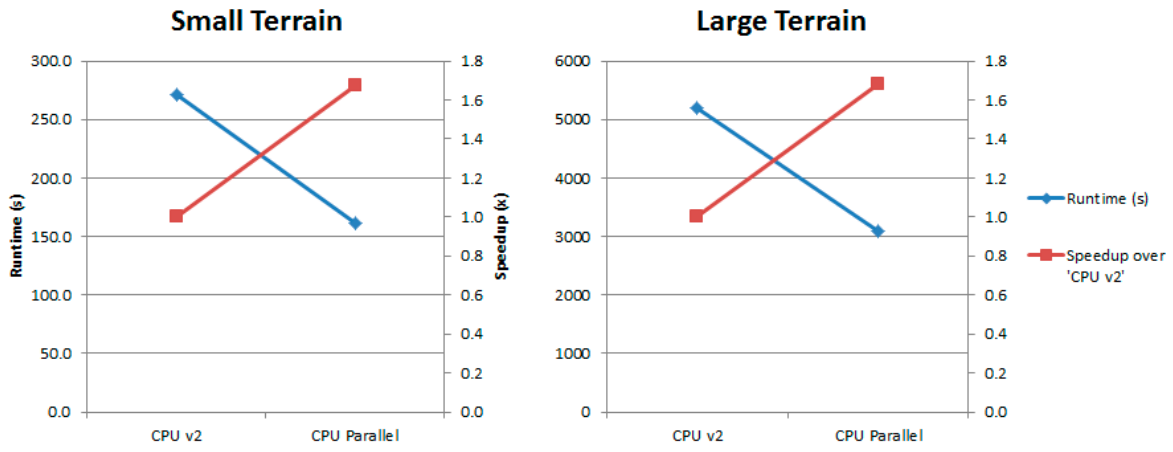


Figure 8.4: Speedup graph comparing the runtime in seconds and the observed speedup for the parallel CPU implementation over CPU v2. We observe a 1.7 times speedup achieved for both test terrains.

8.1.3 CPU versus incremental GPU implementations

Here we provide the results for our eight different GPU implementations and compare them to the multithreaded CPU version (Figure 8.5). We began with our first GPU version (GPU v1), which is simply the multithreaded CPU code translated to execute directly on the GPU and saw the synthesis time reduced by half. Our second (GPU v2) and third (GPU v3) implementations explored adding additional threads and makes use of shared memory to allow the threads to operate independently, with a single thread summing the value at the end. This again saw our synthesis time cut in half, although our third version actually performed slightly worse than the second. This can be attributed to the additional *if* statement used to initialise the memory on the fly as it introduces another step for the algorithm. The fourth version (GPU v4) again adds more threads, which allows an entire loop to be executed in parallel on the system. We first tried providing 32 threads for each of the 4 cost calculation components as this was the maximum defined control points in previous work. This actually performed significantly worse than the prior implementations.

For our advanced GPU implementations we changed the underlying architecture of the system in order to run each of the candidate transformations in a separate thread concurrently. Version five (GPU v5) extends the work done in version three, which doubles the performance due to the increase in concurrency, netting a 14 times speed improvement on the large terrain. Version six (GPU v6) exploits some setup requirements in the cost calculation algorithm to avoid repeatedly calculating the same value. This version performs slightly worse than version five but the difference is insignificant. This is due to the setup overhead for calculating and storing the values up front. Version seven (GPU v7) attempts to bring together the features of both versions five and six, but the results fall between the two prior versions.

Our eighth version (GPU v8) revisits version four as we performed additional profiling and discovered that there were never more than 3 control points required to describe the user and source candidates under a variety of test cases for our given patch size of 64. We then adjusted the number of threads down to 4 which gave a total of 16 threads required (down from the 128 needed before). Running our tests again for version four yielded a modest improvement. We then included the enhancements from version seven and achieved a total speedup of 21 times overall (Figure 8.6) for the large terrain. The speedup achieved for the cost calculation process alone is 44 times faster

than the CPU implementation. Runtime and speedup graphs for all GPU implementations compared against the multithreaded CPU version are provided in Figure 8.6.

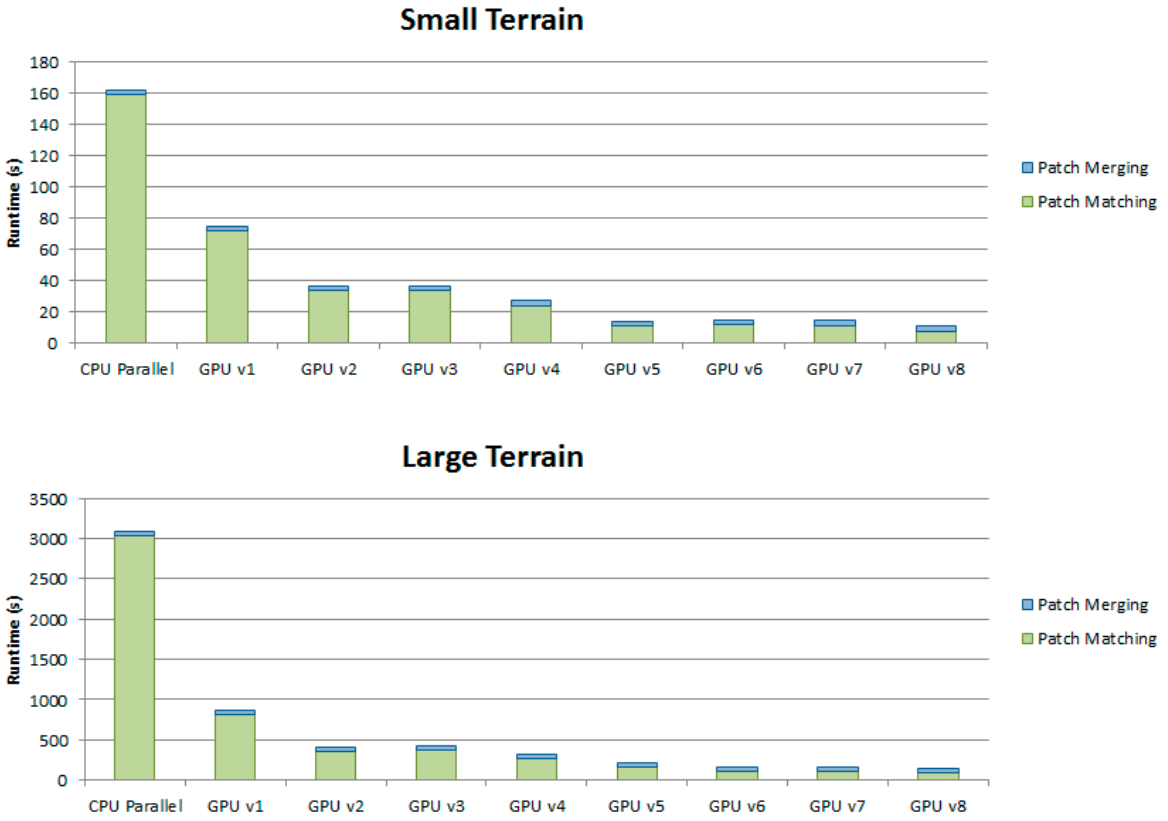


Figure 8.5: Runtime results for the eight GPU implementations compared against the parallel CPU implementation for the small and large terrains. We can see an overall downward trend to the graph with the times decreasing with each iteration. v1 is a translated form of the parallel CPU implementation. v2 adds some shared memory and more threads. v3 attempts to optimise functions but introduces more branching. v4 unrolls an entire loop utilising more concurrent threads. v5 changes the architecture to allow a new dimension of threads for improved concurrency. v6 optimises v5 preventing unnecessary recalculation of values. v7 combines elements from v5 and v6. v8 revisits v4 and incorporates the newer changes in v7. Full runtime values are presented in Table 10.3.

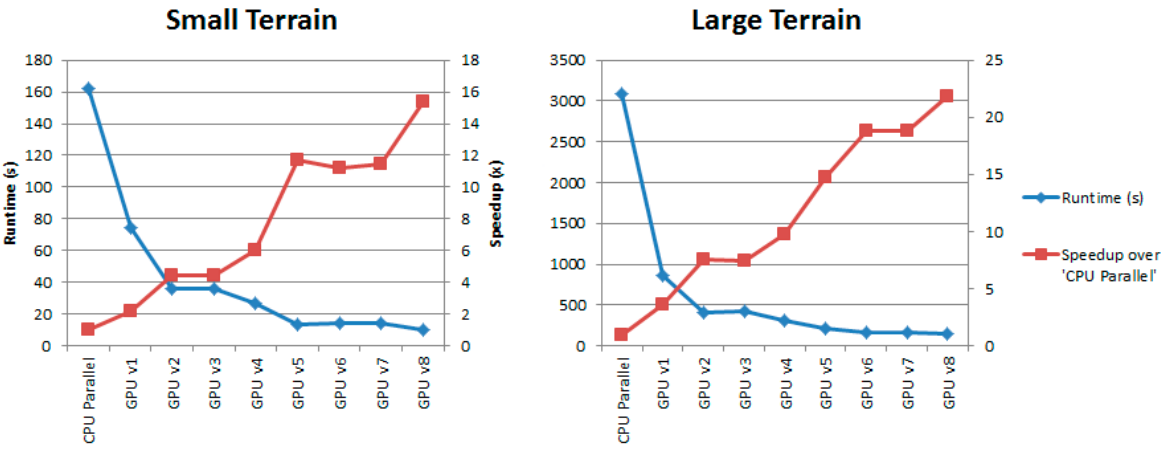


Figure 8.6: Speedup and runtime graph comparing the parallel CPU version against all eight GPU implementations. Similar performance is noted for both the small and large terrains, although a slightly higher speedup is noted for the larger terrain.

8.1.4 Utilising GPU Texture Memory

We added the use of texture memory to our system. This can be combined with any of the aforementioned GPU implementations. Texture memory is part of the GPUs global memory, but is specialised in that it is spatially cached, which allows for quick read access for neighbouring locations. This is useful for our extraction kernels in which we extract pixel data from source images and apply transformations to rotate the candidates. From the results in Figure 8.7, we observe a small increase in performance for our final GPU implementation when using texture memory for both user and source patch extraction. The performance increase is only minor as the existing implementations already make use of coalesced global memory for optimal memory performance. This brings the total speedup over our CPU implementation to 24 times, with the cost calculation stage seeing a 56 times speedup for the large terrain.

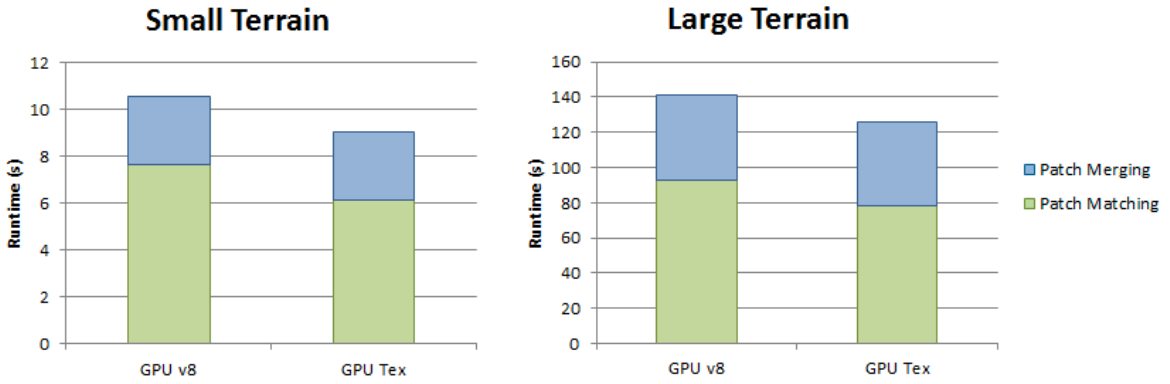


Figure 8.7: Runtime results for our texture memory GPU implementation being compared against GPU v8. There is a slight performance gain when using texture memory. This is because we already are using coalesced memory access for our image data. Full runtime values are presented in Table 10.4.

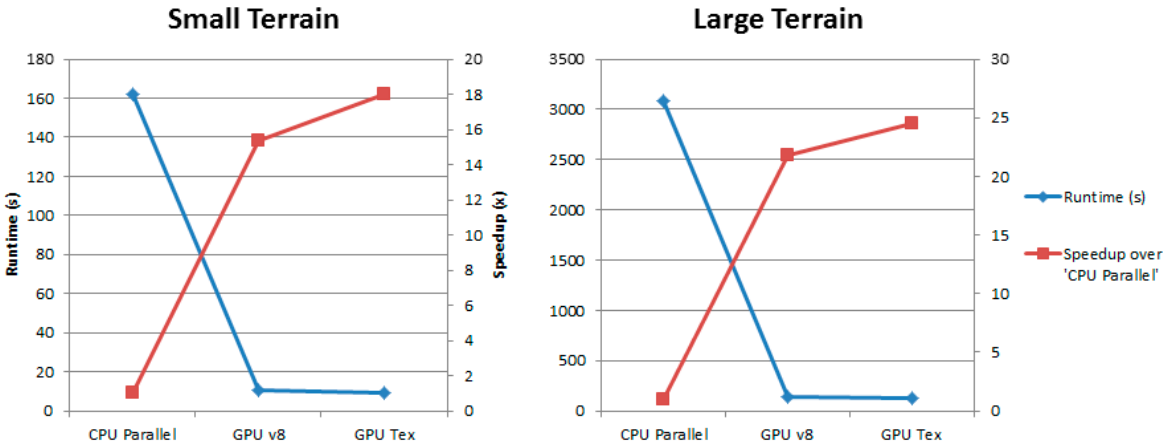


Figure 8.8: Speedup and runtime graph comparing the use of GPU texture memory against the parallel CPU and GPU v8 implementations. Using texture memory now brings the total speedup to 24 times fast than the parallel CPU implementation.

8.1.5 CPU versus GPU Sorting of Candidates

For our GPU implementations we developed three independent variants for sorting of the candidates based on their computed cost values. The above tests were conducted using our own sorting kernel (as described in section 6.4.4). An external library, Thrust (2013), was also used to provide a more hardware optimised sorting solution to improve system performance. We also developed a CPU sorting algorithm that makes use of the C++ stable_sort function (C++, 2015) to compare against the GPU versions. From the results (Figure 8.9), we note that there is a modest improvement by changing the method for candidate sorting. Candidates are sorted after the cost calculation stage, which produces the best subset lists. These are then evaluated during the best patch location stage, which again makes use of a sort to determine the best overall patch. The Thrust version improves the speed of the entire system even further due to the code being highly optimised and purposely designed for the hardware to extract maximum performance. Comparing the speedup of the Thrust version over our previous best GPU implementation with texture memory, we observe a total speedup of 27 times (Figure 8.10) for the large terrain. Thrust sorting will thus be used for subsequent tests covered in this chapter.

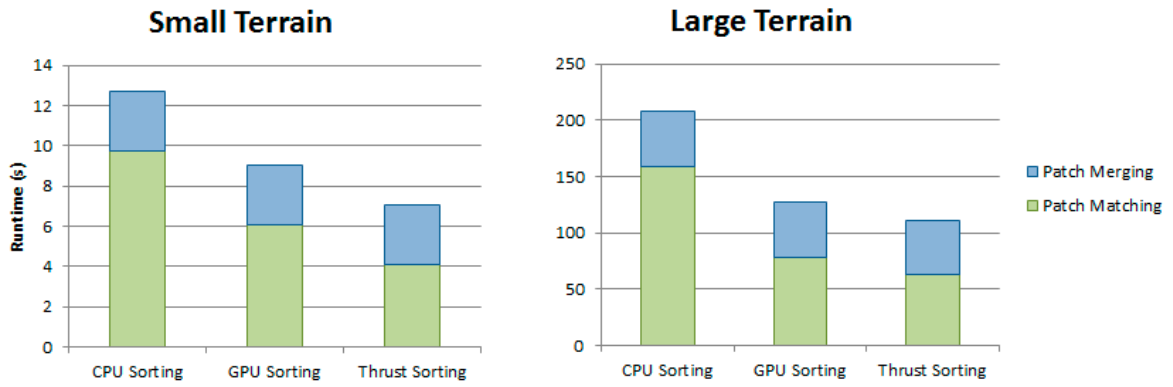


Figure 8.9: Runtime results comparing the three different candidate soring functions. The *Patch Matching* component in the graph includes the sorting operation, which is why we see the green bars decreasing in size with the GPU and Thrust (2013) implementations. Full runtime values are presented in Table 10.5.

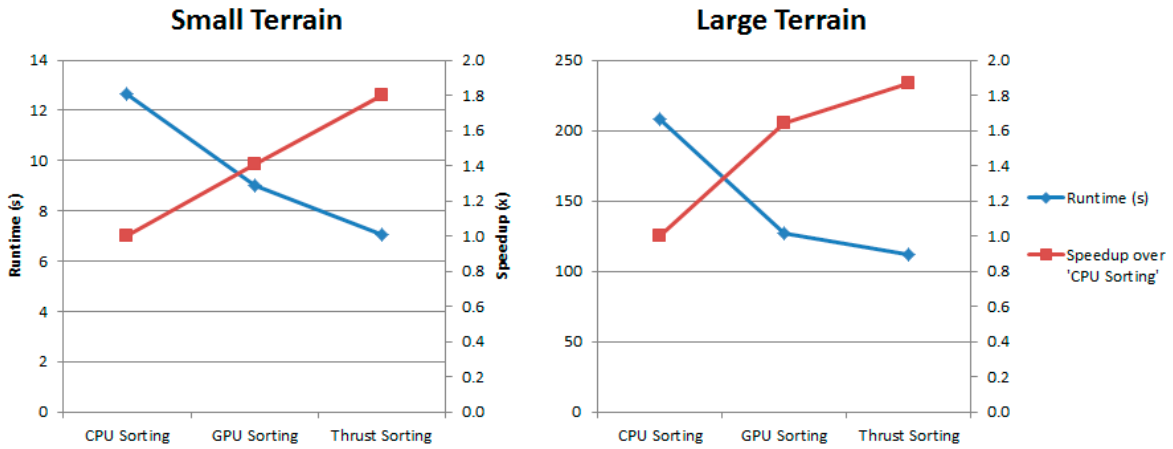


Figure 8.10: Speedup and runtime graph comparing the three different candidate sorting functions. We see a modest performance increase when using the GPU for sorting, even with our simple kernel implementation. Using the Thrust (2013) library further improves the result due to their kernel being highly optimised.

8.1.6 Blocked GPU for Asynchronous Processing

The last optimisation we developed was an asynchronous blocked design that would process the user patches in groups of 50. When one batch is completed, it is sent back to the CPU where it is processed for merging, while the next batch is processed on the GPU. This allows for asynchronous processing by the CPU and GPU to potentially reduce idling.

The results in Figure 8.11 show an impressive reduction in synthesis time compared to our previous best result (GPU v8 with texture memory and Thrust sorting). Examining the timing values in Table 10.6, the results are initially confusing as adding the three main components results in a larger value than the actual runtime. This is a result of the *Source Patch Matching* and *Patch Merging* operations executing asynchronously. When there are less than 50 user patches then there is no benefit to the blocked design. We now observe that our maximum speedup achieved is 48 times (Figure 8.12) faster than our CPU implementation. The speedup is greater when larger terrains are used with a high number of patches to match.

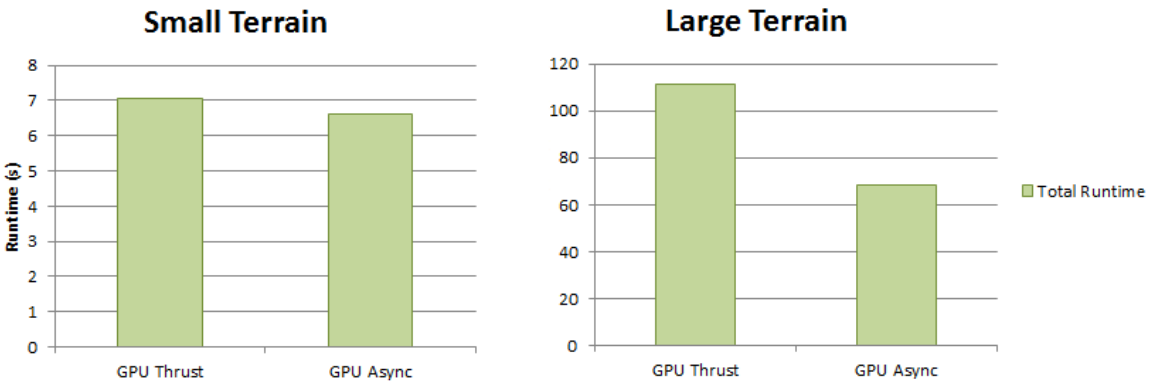


Figure 8.11: Runtime results comparing against our asynchronous blocked design against the current best GPU implementation using Thrust sorting. For this test we need to compare the total runtime as the two components are run concurrently on the CPU and GPU, which reduces the overall time as there is far less idling occurring. The timings for matching and merging are approximately the same but due to running them asynchronously we see a reduced overall runtime (Table 10.6).

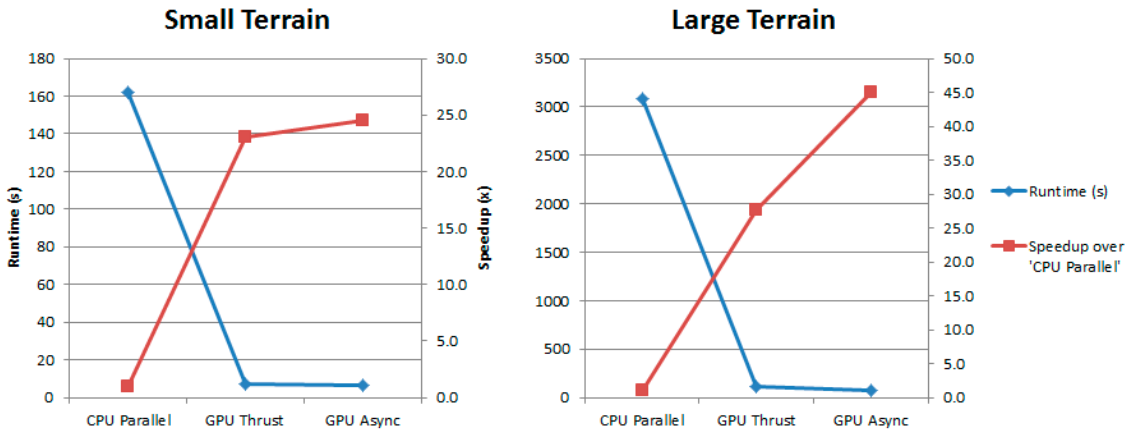


Figure 8.12: Speedup and runtime graph for the asynchronous blocked design against the parallel CPU and Thrust GPU implementations. We see a marginal increase with the asynchronous design for the small terrain with a very large increase on the large terrain. This is attributed to the total number of features, as the large terrain has a high feature count it is divided up into more blocks which enables the concurrent processing on the CPU and GPU.

8.1.7 Culling Nearby User Patches

One optimisation that is included in all our test cases works by culling detected user patches that are too closely spaced to an existing chosen patch. This can occur as an artefact from the feature detection algorithm where branch reduction fails and results in multiple segments that describe the same feature line. However, this issue is not a bug in the implementation but rather an unfortunate feature of the algorithm. This produces multiple patches that describe the same feature and leads to over-synthesising, a wasteful exercise. In order to better test this scenario we designed two new test terrains (Figure 8.13). Table 8.2 records the total number of patches detected from feature synthesis and the number of features used after the culling algorithm.

	Small Terrain		Large Terrain	
	Total	**Culled To**	**Total**	**Culled To**
Ridges	201	88	909	722
Valleys	146	84	753	661

Table 8.2: Number of detected features before and after the culling algorithm. The dimensions for the terrain are, 512×512 for the small terrain and 5000×5000 for the large terrain.

We note the speedup obtained by removing the unnecessary patches in Figure 8.15 and Figure 8.16: there is a noticeable decrease in synthesis time after reducing the number of user patches. This is less apparent on the larger terrain where the proportion of overlapping patches is far less. The overlap issue is exacerbated along 45° angles or if the drawn lines are too thick, as long parallel paths form that lead to divergence on the same feature (Figure 8.14).

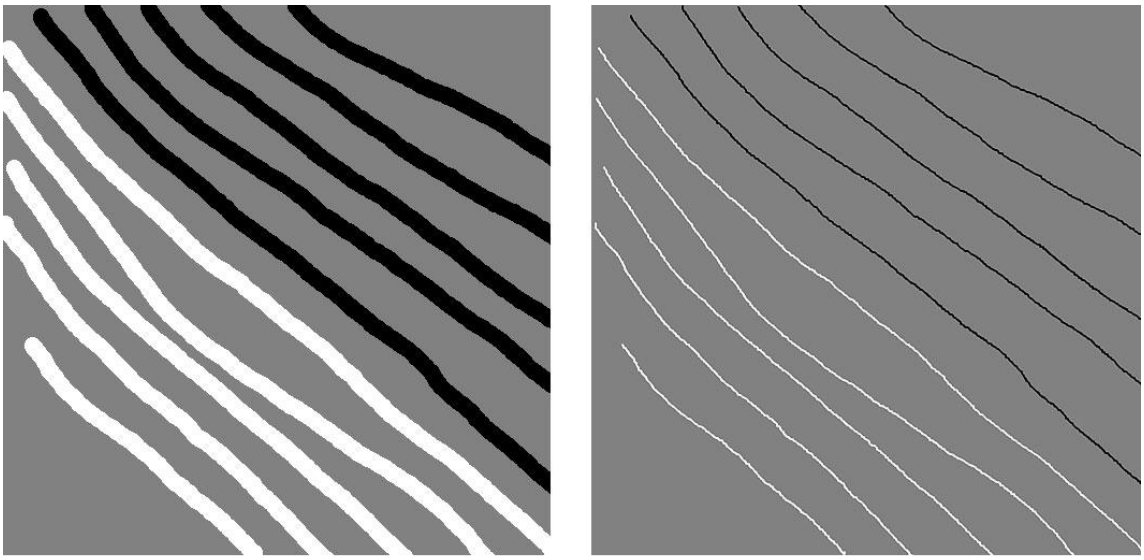


Figure 8.13: The two test images used to test culling of excess user patches. These were designed to exacerbate the unfortunate feature of the original feature extraction algorithm. a) The small 512×512 terrain. b) The large 5000×5000 terrain. The white lines represent ridges with the black lines being valleys as detected by the system.

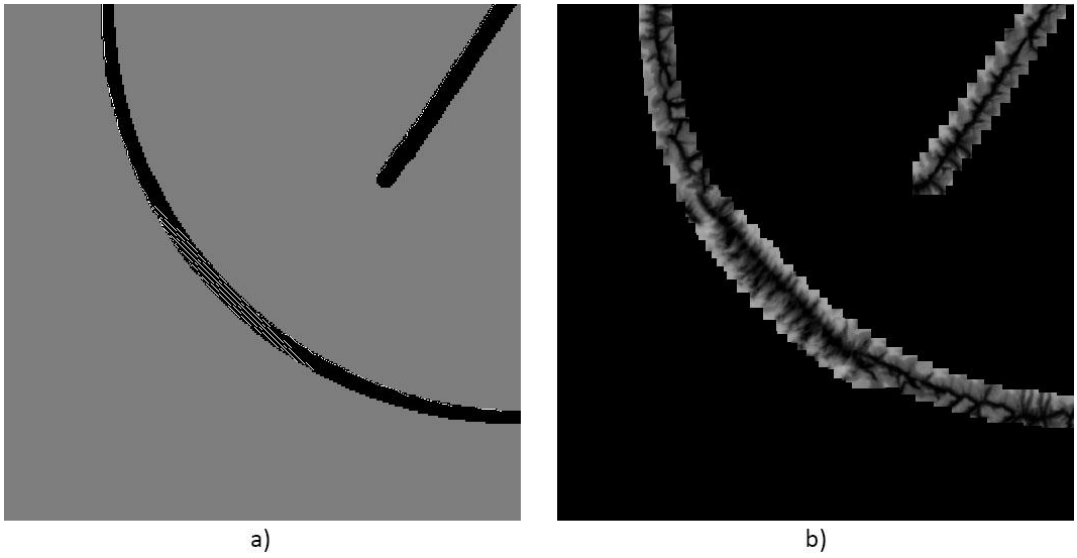


Figure 8.14: (a) Example of error with feature detection engine forming multiple parallel lines. (b) This results in heavy overlaying of patches, which wastes performance. These excess patches are culled by the system.

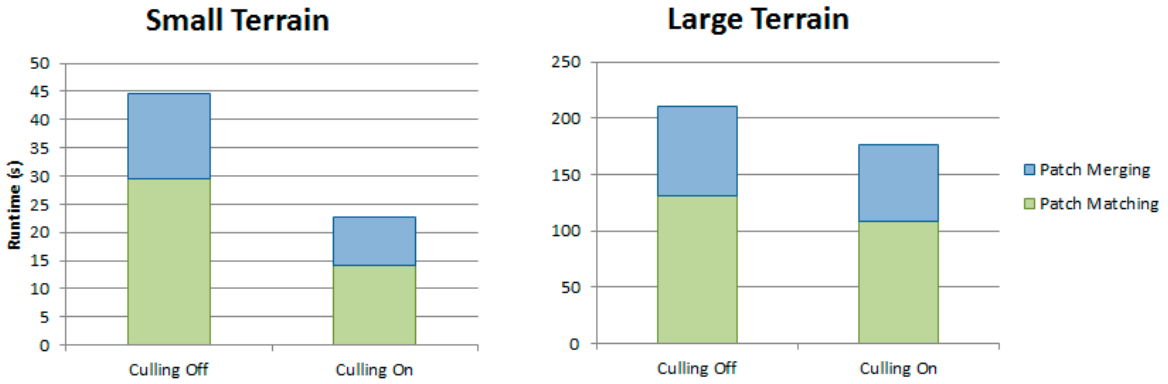


Figure 8.15: Runtime results comparing the implementations when either culling of nearby user patches or not. This is an issue with the original feature extraction algorithm. We address this by examining user patches and removing those that are in close proximity to one another. This reduces the total number of features requiring synthesis and thus improves performance as shown above. Full runtime values are presented in Table 10.7.

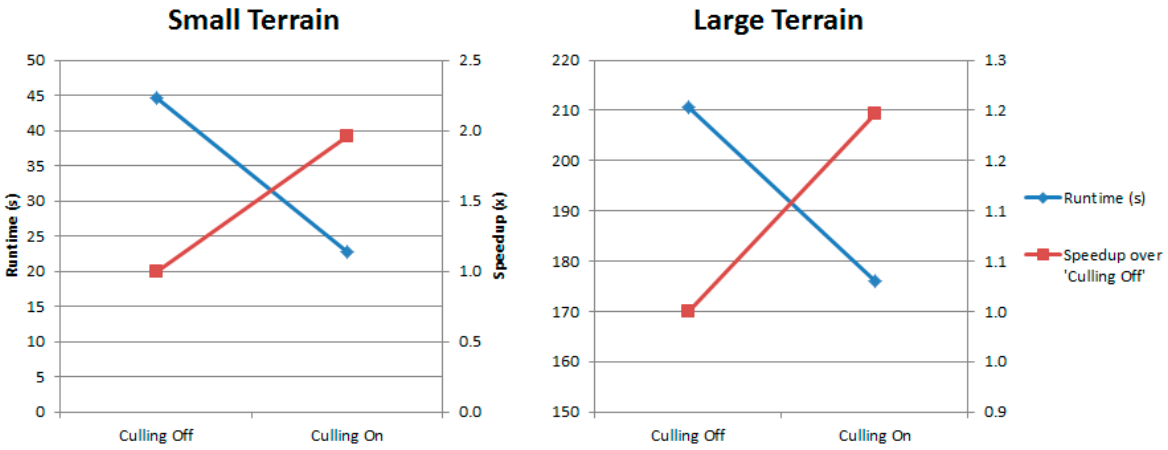


Figure 8.16: Speedup and runtime graph showing the performance gain when culling nearby user patches that are not required. We see a higher gain in the smaller terrain as the proportion of culled patches is higher than the larger terrain.

8.1.8 Feature Complexity Change

In order to evaluate the scalability of our system we developed a test that progressively increases the total features being matched against. We achieved this by sketching the initial test image and then duplicating the strokes around the image to roughly increase the complexity at a fixed interval. This allows us to predict how our system will work given more complex user sketches. The results in Figure 8.17 can be better visualised as a graph (Figure 8.18), where we compare the runtime and speedup against the number of features. As the features increase linearly so does the synthesis time. Since the system scales linearly, larger terrains can be produced within an acceptable amount of time.

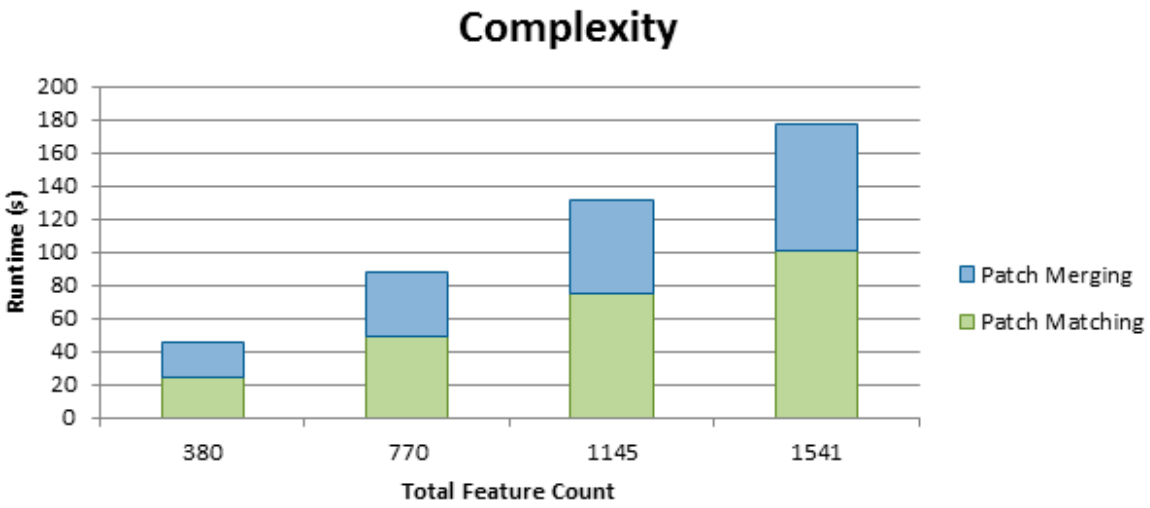


Figure 8.17: Runtime results for complexity with increasing total number of patches requiring synthesis. We observe that with an increase in the number of features we see an increase in the time required, with approximately the same proportion of time spent on matching and merging components. Full runtime values are presented in Table 10.8.

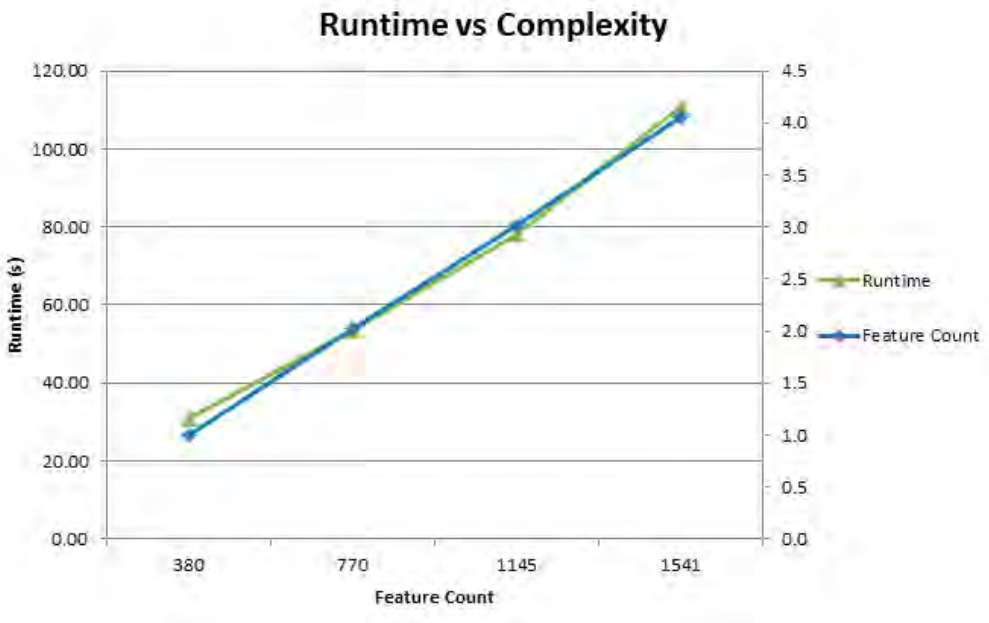


Figure 8.18: Plotting the runtime and feature count values on a graph shows a linear relationship for both, which indicates that the system scales well when increasing the number of features.

8.2 Non-Feature Synthesis

The non-feature synthesis GPU-based component was only partially completed due to scope constraints. Nonetheless, we did implement an initial GPU-enhanced version that borrows techniques from our feature synthesis implementations to perform the cost calculations on the GPU. The rest of the processing remains single threaded and CPU-bound. Figure 8.19 presents results, comparing our CPU and GPU enhanced implementations. We observe a very large speedup in the cost calculation stage but no noticeable speedup in other aspects (Table 10.9). More research time is required to explore moving other components to the GPU. For instance, the selection of the next target; this requires a large proportion of the overall runtime, especially on larger terrains where the number of *holes* in the terrain increases dramatically. Given the very long runtime for non-feature synthesis, especially on the large terrain, any speedup will have significant impact on the total time. The GPU-version reduces the synthesis time from over 90 minutes to just 54 minutes.

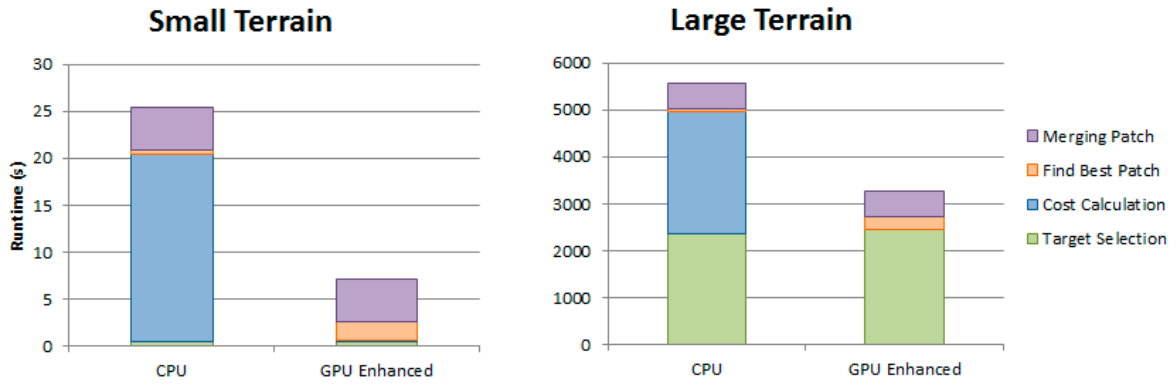


Figure 8.19: Runtimes for the four main contributing components during non-feature synthesis comparing a CPU only implementation to a GPU-enhanced one. We observe that calculating the candidate costs on the GPU significantly reduces the required time. Examining the time values in Table 10.9 we see a 200 times speedup for cost calculation on the small terrain.

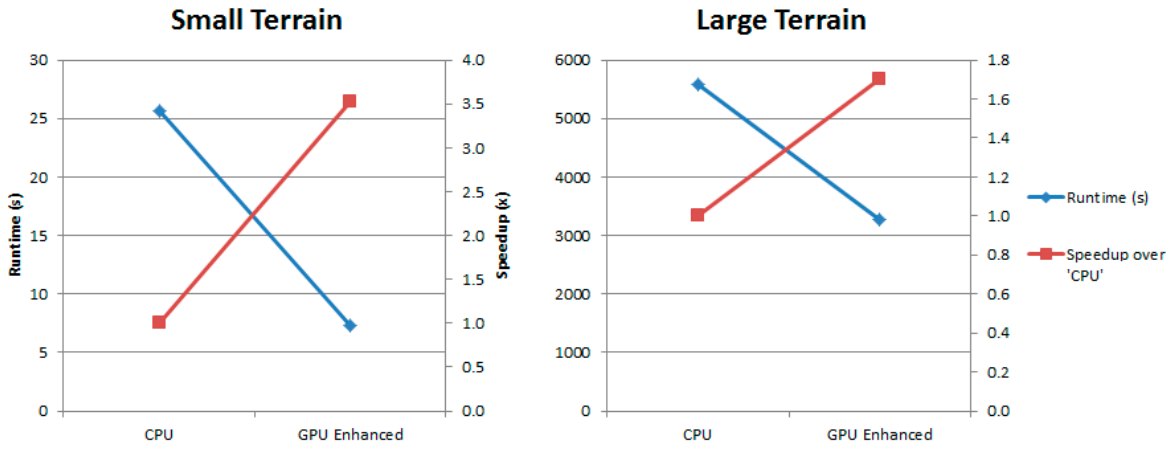


Figure 8.20: Speedup and runtime graph for the non-feature synthesis stage of our system comparing CPU bound and GPU-enhanced implementations.

8.3 Full Synthesis

The tests covered in this section seek to quantify system aspects that globally affect both feature and non-feature synthesis, such as changing the patch size. We also compare our system with the previous work by Tasse et al. (2011). For these tests, we use our best performing implementation: GPU version eight with texture memory, Thrust sorting and asynchronous blocking.

8.3.1 Comparison with previous work

We compare our system to Tasse et al. (2011). We were able to obtain a copy of their software system in order to do a direct comparison with our system on the same hardware. Unfortunately only their CPU implementation executed correctly. We include our CPU version to more closely compare results and draw comparisons from the results section of their paper. For the purposes of the test we configured our system to only use a single source file. Another issue encountered was their system was unable to handle our large test image, which we then substituted for a different smaller image with dimensions 2000×2000 (Figure 8.21).

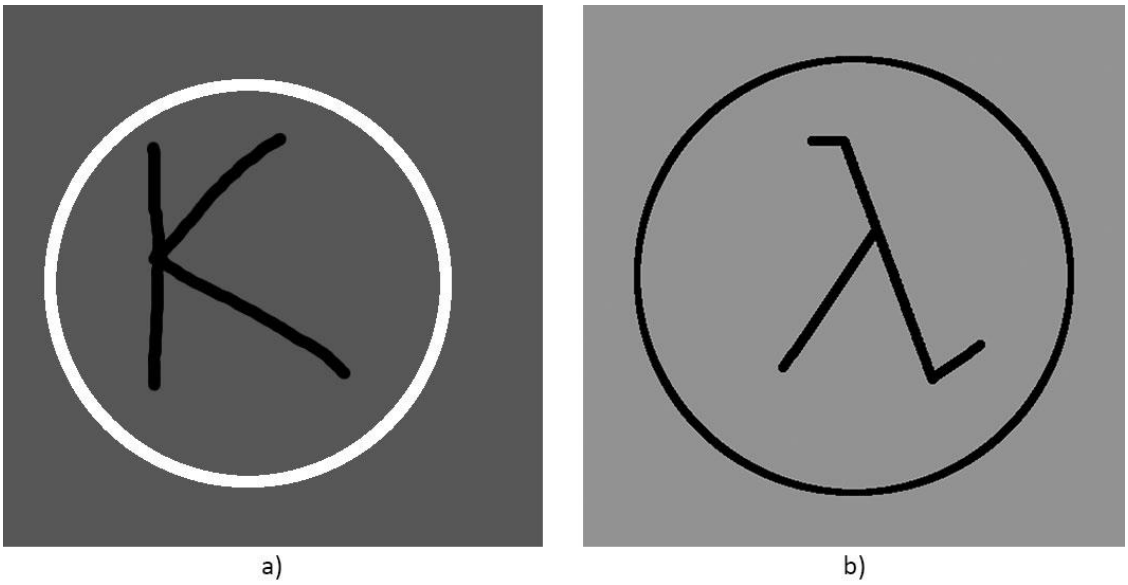


Figure 8.21: The user images used for this test. a) The original small 512×512 terrain. b) The larger 2000×2000 image, which only features valley data.

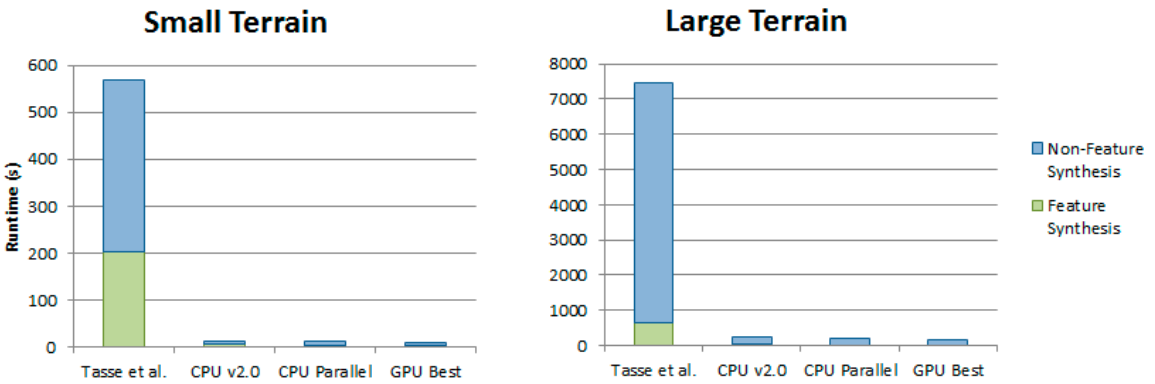


Figure 8.22: Runtime results comparing the previous work by Tasse et al. (2011) to our system. We were only able to run their CPU version, which is why we include our two CPU implementations and our best GPU implementation. The graph above shows that the runtime for our system is far less with the three implementations appearing as tiny columns. Table 10.10 provides the actual runtime values, which better shows the time difference between all the versions.

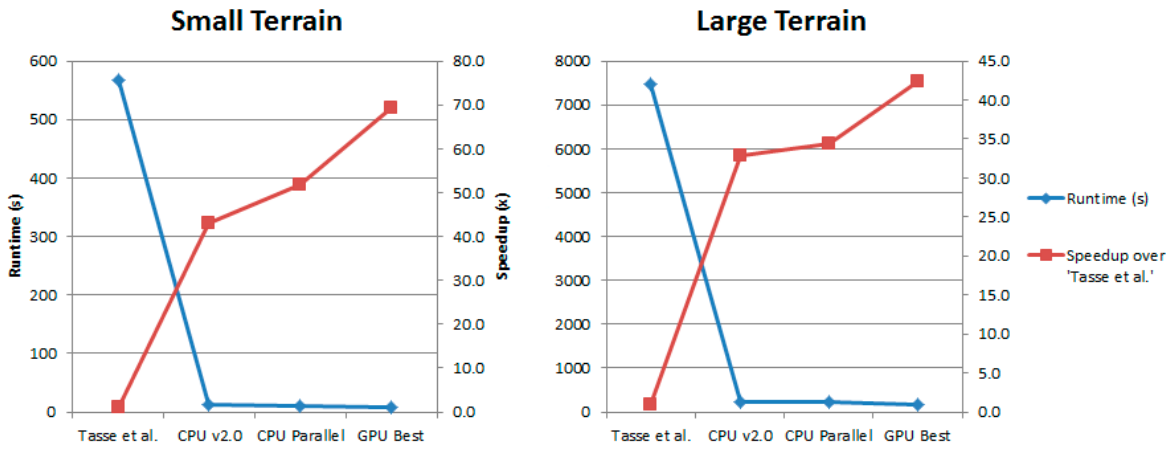


Figure 8.23: Speedup and runtime graph comparing the previous work to our system. Here we see the large performance increase our system achieves when running under the same test conditions.

From the results in Figure 8.22 and Figure 8.23, we observe that our system, even when using multiple source files, runs significantly faster than the previous work. This is not entirely a fair test, however, as we were unable to compare our system to their GPU implementation. Examining their results section we extrapolate that their system would yield faster results when conducting a full synthesis, as the entire system was run on a GPU. Our system lacks acceleration in the non-feature synthesis area, with most of this processing executed on the CPU. However, the advances we have made for our feature synthesis component are significantly faster. The resulting terrain produced by their system appears similar to our own output, with both matching the input user sketch. However, based on the test sketch, the image synthesised with our system appears clearer, with bolder feature traits and also follows the sketch better.

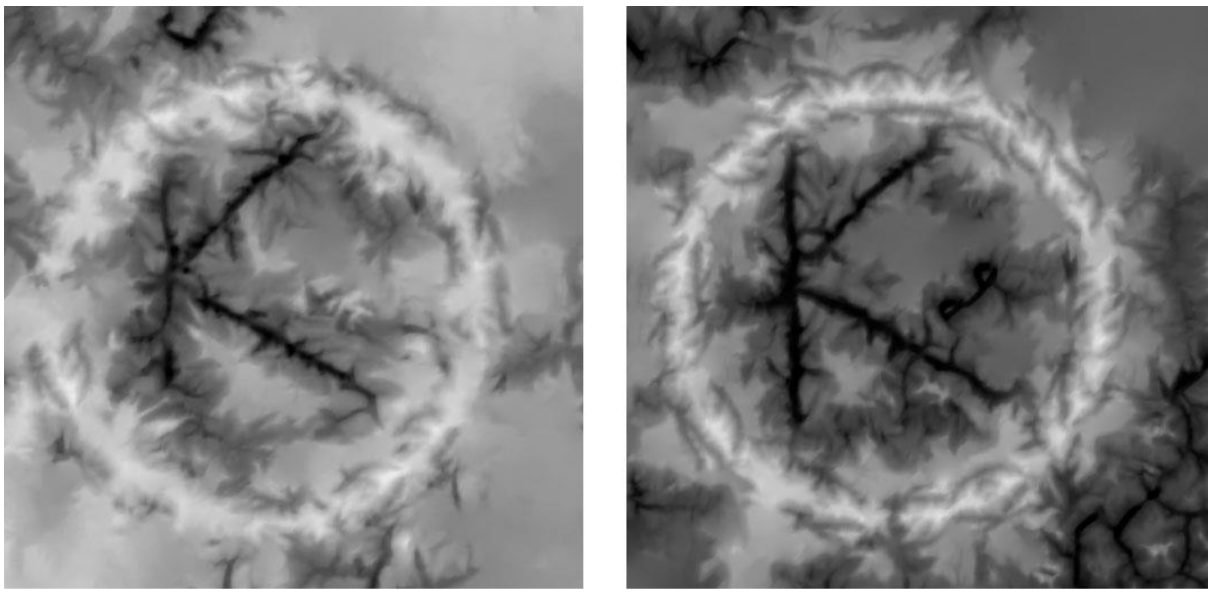


Figure 8.24: a) Output from Tasse et al. (2011) system. b) Output from our system using the same single source file.

8.3.2 Single versus Multi-Source synthesis

A core objective of our research was to incorporate the use of multiple input source files to provide a greater candidate pool and improve the overall variety. We compare our synthesis result when given a single source file and when given a collection of fifteen varying source terrains (Figure 8.25). As expected we see a decrease in performance with the system requiring twice as long for the small terrain to execute. The feature matching stage is where most of the processing time is spent, as there are many more candidates to evaluate for each user patch. The results are similar for the large terrain with noting a decrease in performance. However, since we are making use of the asynchronous blocked implementation, the cost calculation and patch merging steps are run concurrently on the large terrain. This reduces the impact of using multiple source files compared to the small terrain. This occurs on the larger terrain only as the number of features requiring synthesis is high enough to have the system split it into blocks.

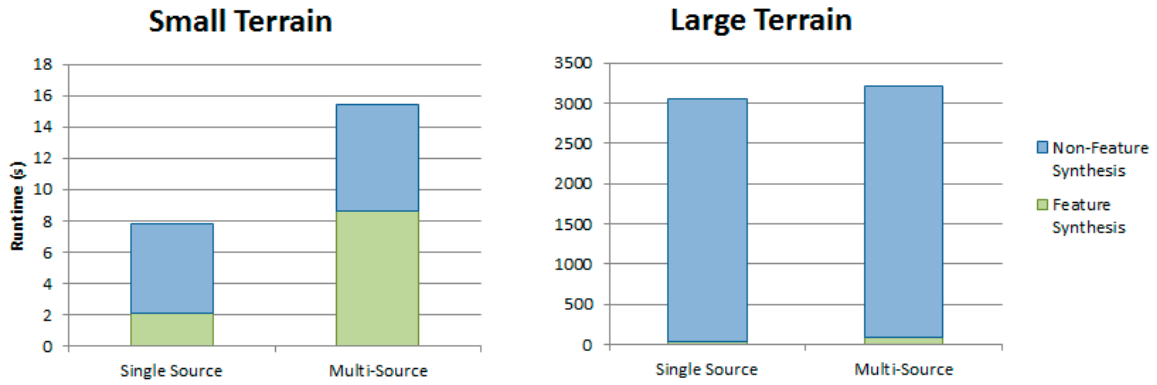
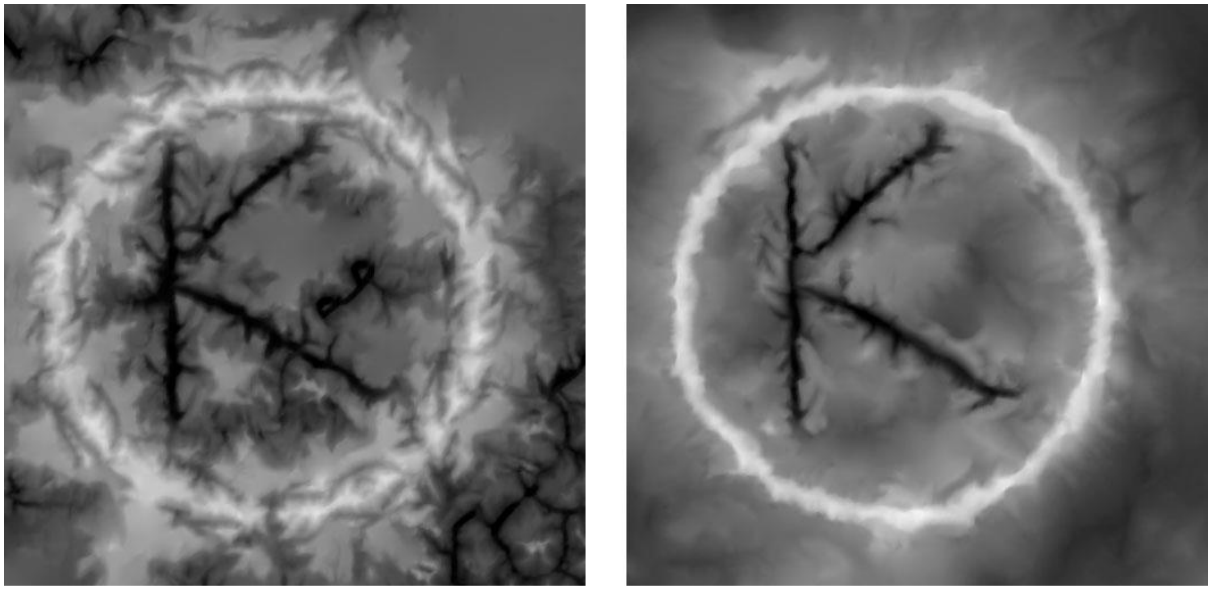


Figure 8.25: Runtime results when running either a single or database of fifteen source files. The figure shows the times for the feature and non-feature synthesis components. We see the majority of the impact being confined to the feature synthesis stage, this is due to there being more candidates needing evaluation. Non-feature synthesis results are very close in size as there is more of an impact from the number of iterations required to fill the output terrain with the candidate matching only being a small percentage of the runtime. Full runtime values are presented in Table 10.11.

When examining the non-feature synthesis results, we note that the number of source files has a much smaller impact on the relative synthesis times for the small and large terrains. The slowdown occurs with the generation of the source candidates and when sorting, due to there being a larger number of candidates. Target selection is one of the largest components leading to the high cost for non-feature synthesis. It is unaffected by the number of candidates being evaluated. As such larger terrains are less influenced by the size of the source database. Figure 8.26 shows the small terrain after synthesis when using a single and multiple source files, with the multi-source output having better ridge data. Another use case that our multi-source system would excel in is, when the user is attempting to generate a mountainous landscape, for instance, but the single source file is based off the Grand Canyon without sufficient mountainous data (Figure 8.27). The system would produce poor results due to the lack of diversity, which is resolved by using various terrain types as input sources.



a) b)

Figure 8.26: Output terrain for: a) Single source. b) Multiple sources

a) b)

Figure 8.27: Example when running a ridge only terrain using a) Single source – Grand Canyon. b) Multiple sources. The single source does not have sufficient ridge data resulting in a poor terrain compared to the clearly defined structure when using multiple sources.

8.3.3 Patch Size change

One final test examines the effect of varying the size of patches. This alters all aspects of the system, as a reduction in patch size increases the number of user patches generated to cover the same sketched area. As discussed earlier, we chose a base patch size of 64×64, which was based on CUDA grouping 32 threads into warp units. We tested patch sizes in increments of 32 starting at 32×32, up to 160×160. Our results are presented in Figure 8.28, with the runtime and speedup visualised in Figure 8.29.

An interesting pattern emerges when examining performance results for the feature and non-feature components. For feature synthesis we see an improvement in performance when moving from 32×32 to 64×64, but the performance decreases thereafter as the patch size increases. This is explained by the increase in patch area, which impacts the performance when calculating the cost for a patch. This is largely mitigated by the decrease in the number of candidates to synthesise. However, when it comes to patch merging the increased patch area impacts the performance more severely, taking up more of the synthesis time as the patch size increases. These results are mirrored for both the small and large terrains, which leads us to conclude that the influence of patch size is independent of the size of the synthesised terrain.

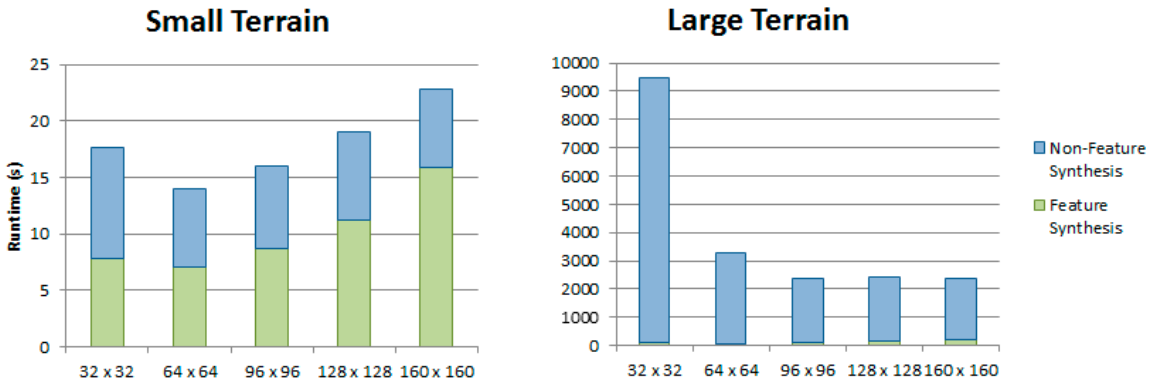


Figure 8.28: Runtime results when using different patch sizes to synthesise terrains. We observe that for the small terrain the optimal patch size is 64×64 with the large terrain performing better with larger patch sizes. Upon further inspection of the timing values (Table 10.12), we note that for both terrain sizes the feature matching component performs fastest with a patch size of 64×64. Larger patch sizes reduce the non-feature synthesis time as more data is placed on each iteration, requiring less overall.

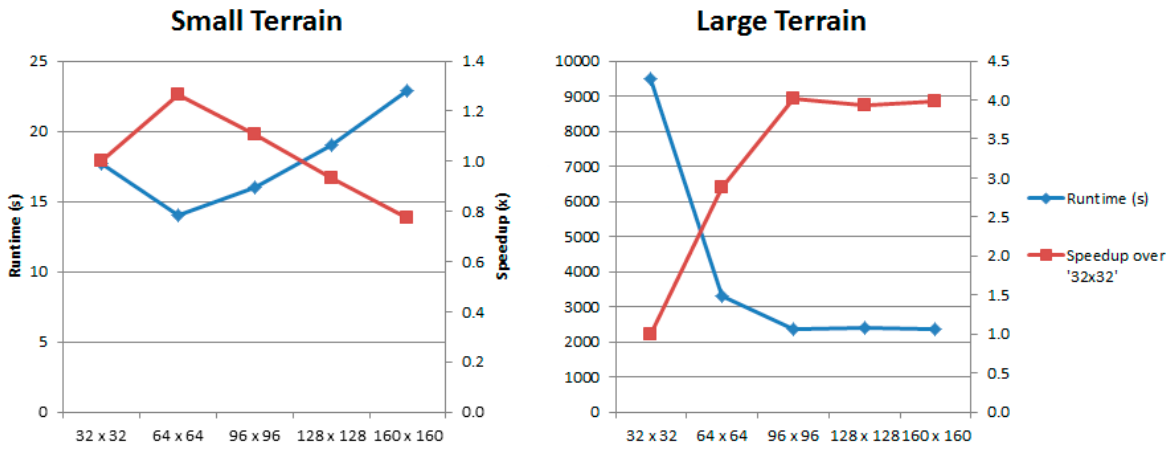


Figure 8.29: Speedup and runtime graph showing the effect of varying the patch size for synthesis operations. For the small terrain the optimal size is 64×64, with the large terrain performing best with the 96×96 patch size.

Turning to the non-feature synthesis results, we see an increase in the performance as the patch size increases. This performance gain is attributed to the larger area covered by each patch placement operation. With a larger area being merged into the final terrain, there are significantly fewer open areas. This reduces the number of operations required to complete the synthesis and thus the time required. This pattern is exhibited for both terrain sizes. However, the large terrain requires a very large number of patches to complete synthesis and the time reduction is relatively significant. Similarly, when the patch size increases the cost calculation step is reduced, as is the selection of the next location to synthesise, due to reduced number of iterations required overall. We observe an increase in the merging time due to a larger patch area having to be evaluated when running the complex merging algorithms.

The above results motivate our choice for the patch size of 64×64 especially with the feature synthesis stage. This size also ensures the highest relative speedup between successive patch sizes.

8.4 Summary

Here we evaluated the performance of all the various implementations of our system. We started with a primitive CPU implementation and progressed to an asynchronous blocked design that balances work between the CPU and GPU in order to maximise performance. We demonstrated a peak speedup of 45 times, when looking at feature matching and merging alone, and a 2.7 times speedup when including non-feature synthesis. We also evaluate our system against the previous work of Tasse et al. (2011) and observe speedups of 42 times over their CPU implementation for a large test terrain. Unfortunately we did not have access to their GPU implementation in order to do a comparative test but speculated on the results. Our inclusion of a multi-source database is shown to significantly improve the output terrain, especially when a single source lacks the feature types in the user sketch. Our system only parallelises a small portion of the algorithm and obtains the high speed increases; Chapter 0 presents some aspects that could be explored to further increase performance. We now showcase some example terrains with 3D renderings of the result.

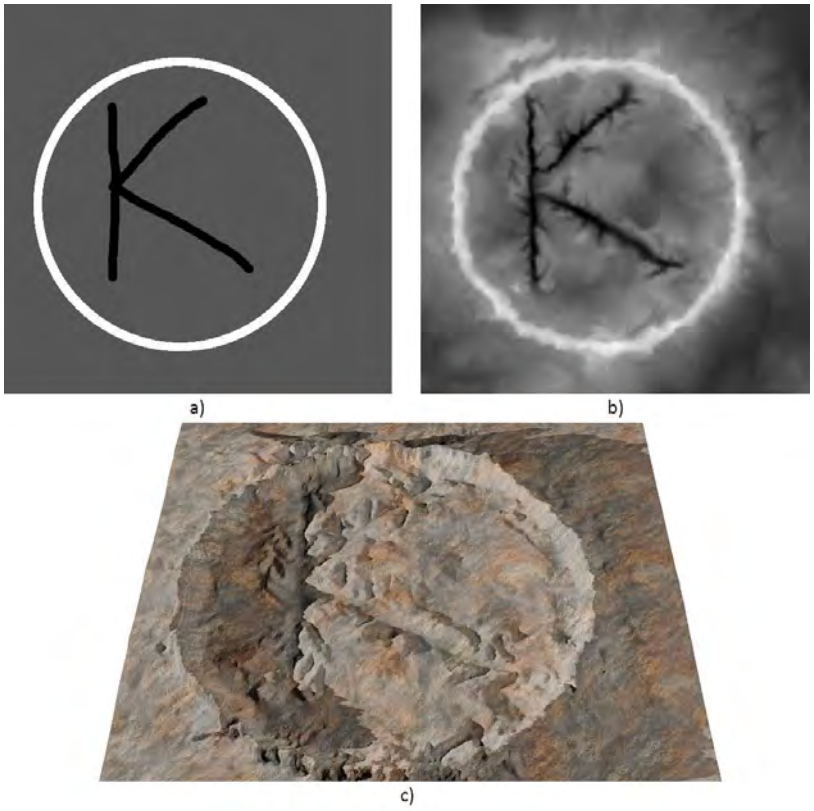


Figure 8.30: Our small test terrain (512×512). b) The output from our synthesis system (Completed in 13 seconds). c) 3D rendering of the terrain.

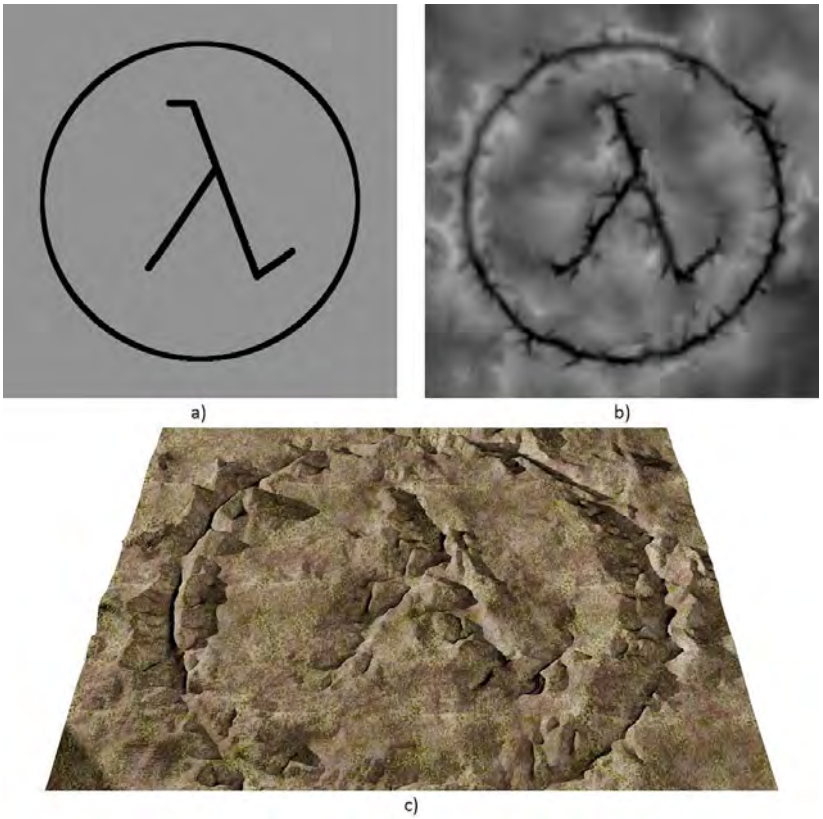


Figure 8.31: a) The lambda symbol drawn as valleys (500×500). b) The output from our synthesis system (Completed in 14 seconds). c) 3D rendering of the terrain.

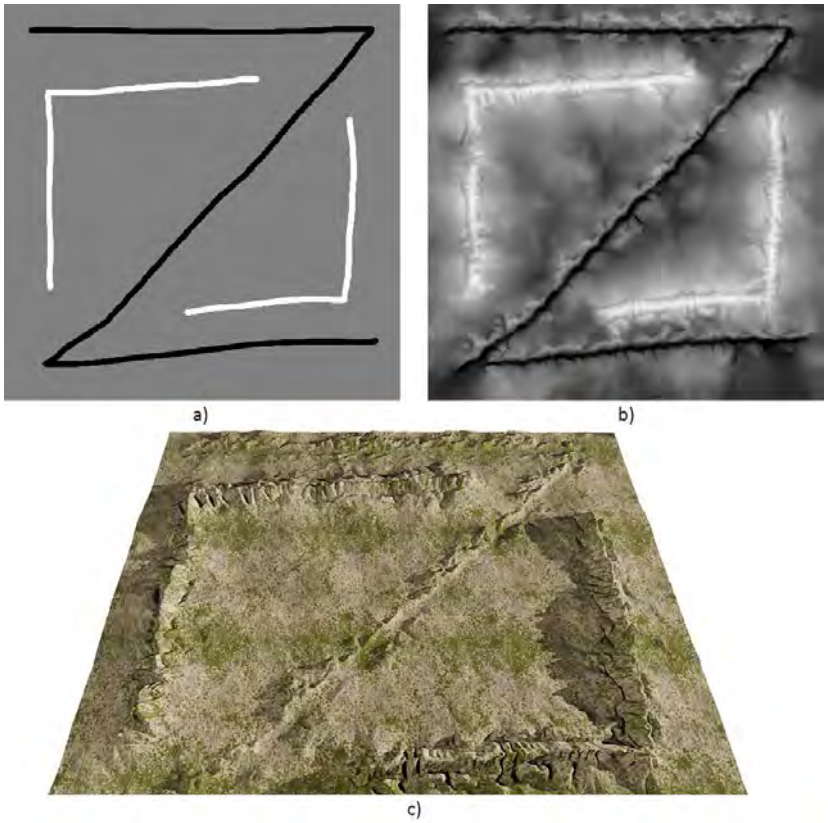


Figure 8.32: a) A combination of ridges and valleys (1000×1000). b) The output from our synthesis system (Completed in 52 seconds. c) 3D rendering of the terrain.

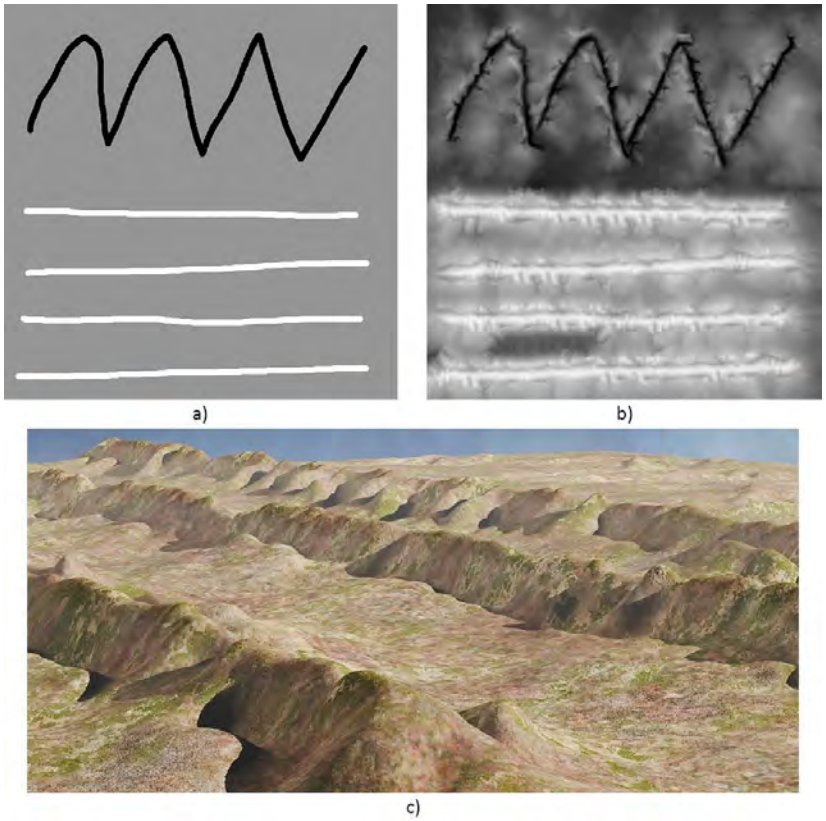


Figure 8.33: a) A combination of ridges and valleys (1000×1000). b) The output from our synthesis system (Completed in 49 seconds. c) 3D rendering of the terrain.

9 Conclusion

The primary objective of our research is to build a terrain synthesis system that is capable of rapidly generating realistic terrains from a simple user-sketched image. We base our research on the work by Tasse et al. (2011) as their work produces highly realistic terrains using patch-based techniques. We develop several extensions to their research including the use of multiple source terrains in an effort to increase the candidate pool available during synthesis to allow for more varied generated terrains. We also develop a highly parallelised GPU solution to dramatically accelerate the synthesis operation.

We developed a number of different implementations ranging from a simple CPU only one, to an advanced asynchronous design utilising both the CPU and GPU for maximum performance. The inclusion of multiple input sources addresses limitations where certain types of user feature would not be possible to synthesise on a single source due to limited variability within it. This vast increase in candidates to choose from results in better matching features being synthesised. Other optimisations were implemented during feature matching, which results in terrains with clearer, bolder feature traits that more closely follow the user's original sketch.

From tests, we determine that our system is capable of producing terrains that match the quality of the previous system (Tasse et al., 2011) due to using components of their existing system and integrating the new components. We were able to produce more diverse terrains through the use of multiple input sources. The performance impact when using multiple sources varied between 0.5 times to 0.95 times the original synthesis time, depending on the size of the terrain being generated. This was for our test system with fifteen source files and further increasing this would change the results and requiring more synthesis time. Several optimisations were integrated, which further improved performance. Our highest speedup obtained for a hybrid CPU/GPU solution overall was 76 times faster than the first implemented CPU version. This reduced the feature synthesis time for small terrains from 272 seconds to just under 7 seconds and from 90 minutes to just under 70 seconds for our large test terrain. Comparing our system to the previous work we see a significant increase in performance, despite not having ported our entire system to be GPU based.

Revisiting our comparison of synthesis techniques from section 2.4, we saw that texture-based methods have a slow speed in comparison to fractal and physics-based techniques (Table 9.1). However, our system shows large speed improvements, reducing synthesis to just seconds while offering a high degree of user-control and realism. Fractal based methods can achieve interactive synthesis and allow users to immediately see the effects of parameter changing but do not have the degree of control our system provides. We describe the main limitation and some possible future improvements to our system next, but even with these improvements it is unlikely the system could attain interactive speeds due to the very large amount of data that needs to be processed.

	Speed	User-Control	Realism	Main Limitations
Fractal-based	Very fast	Low – High*	Low	• Absence of natural erosion • Non-intuitive control parameters • Pseudo-random output terrain
Physics-based	**Thermal:** Fast **Hydraulic:** Slow	Low	**Thermal:** Medium **Hydraulic:** High	• Complex to implement • Requires a base terrain • Minimal user control
Texture-based	Slow	Medium	High	• Limited user control • Output dependant on number of input terrains (exemplars)

Table 9.1: Comparison of terrain generation methods. Table from section 2.4

9.1 Limitations

Due to the large scope of the project, and the time constraints imposed by a master's degree, we were unable to complete all the desired tasks. The main limitation is that the system was not ported to the GPU in its entirety with the final merging process still being CPU-bound. The merging process is inherently sequential and does not map efficiently to the GPU. We have proposed some solutions to the problem (see below), which would allow for some aspects of the process to be parallelised, but did not have sufficient time to successfully implement them. Another limitation is that the sketching interface only allows for two-dimensional manipulations and has no method for specifying the height of the output terrain, apart from choosing between ridges and valleys.

9.2 Future-work

Our research can be extended by implementing a higher degree of CPU multithreading for the CPU-bound processes to fully utilise the CPU's performance. Closer interoperability between the CPU and GPU would greatly reduce the time either spent idle. Ensuring that maximum performance of the system is reached. Many aspects of the current CPU code could easily be modified to allow for a threaded environment. However, this is not suited to GPU calculation, since the size of the data and the consequent transfer time would far outweigh the computational benefits.

Fully porting the entire system to the GPU would provide greater performance gains, but the caveat is that some aspects might not translate well to a GPU solution and may actually reduce the overall performance. Strandmark and Kahl (2010) developed a distributed graph-cut algorithm, but even their system does not guarantee a speedup. Their work could be built upon to enhance our system. Even if no speedup is achieved, the system would still be capable of processing larger datasets that would not entirely fit in memory.

The core of the synthesis engine is broken up into smaller parallelisable tasks, which are carried out on a single GPU. This system could be extended to support multiple GPU devices, which could dramatically increase the performance, with the only foreseeable issue being the complexity of logic required to control the division of labour between multiple devices. The CPU is often idle during the GPU calculation stages and is thus well suited to handling the control logic.

Improvements to the sketching interface can be made to allow the user to manipulate the strength (height) of the features to be synthesised. This would enable far greater control over the synthesis process. Another addition would be to give the user a method of describing the non-feature areas. This could be achieved by making use of noise profiles of the source terrains. A set of brushes could be provided that the user could paint on the sketch to indicate the type of noise profile to be synthesised. The type of brushes provided could be auto-generated based on the diversity of the input sources provided.

The current system will process all the source files in the specified directory during a synthesis operation. This will not always be the most optimal set, as some terrains might not contain the correct type of features required by the user's sketch. A better system would incorporate a way to categorise the sources based on their features. This would reduce the amount of files queried during synthesis, which would improve the performance and allow for larger databases to be used.

A similar process could be applied to non-feature synthesis to allow the user to control the type of data that is placed in certain areas through an improved sketching interface. Non-feature data could be classified based on noise profiles for the candidate patches, with similar candidates being grouped together into categories. A number of brushes could be made available on the interface corresponding to these various categories. The result of this process is that only candidates from the specified category would be queried during the synthesis for a particular area, which would reduce the number of comparisons. This would also give the user more control over the appearance of the resulting terrain.

List of References

ABDELGUERFI, M., WYNNE, C., COOPER, E. & ROY, L. 1998. Representation of 3-D elevation in terrain databases using hierarchical triangulated irregular networks: a comparative analysis. *International Journal of Geographical Information Science,* 12**,** 853-873.

ANH, N. H., SOURIN, A. & ASWANI, P. Physically Based Hydraulic Erosion Simulation on Graphics Processing Unit. Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia, 2007 New York, NY, USA. ACM, 257-264.

ATI. 2013. *Stream* [Online]. Available: http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/ati-stream-software-development-kit-sdk-v1-4-beta/.

AVATAR 2009. Avatar.

BANGAY, S., DE BRUYN, D. & GLASS, K. 2010. Minimum spanning trees for valley and ridge characterization in digital elevation maps. *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa.* Franschhoek, South Africa: ACM.

BELL, N., YU, Y. & MUCHA, P. J. 2005. Particle-based simulation of granular materials. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation.* Los Angeles, California: ACM.

BENEŠ, B. & FORSBACH, R. Layered data representation for visual simulation of terrain erosion. Computer Graphics, Spring Conference on, 2001., 2001. 80-86.

BENEŠ, B., TĚŠÍNSKÝ, V., HORNYŠ, J. & BHATIA, S. K. 2006. Hydraulic erosion. *Computer Animation and Virtual Worlds,* 17**,** 99-108.

BROSZ, J., SAMAVATI, F. & SOUSA, M. 2007. Terrain Synthesis By-Example. *In:* BRAZ, J., RANCHORDAS, A., ARAÚJO, H. & JORGE, J. (eds.) *Advances in Computer Graphics and Computer Vision.* Springer Berlin Heidelberg.

BRYCE. 2013. *Bryce 7* [Online]. Available: http://www.daz3d.com/products/bryce/bryce-what-is-bryce.

C++. 2015. *C++ stable_sort* [Online]. Available: http://www.cplusplus.com/reference/algorithm/stable_sort/.

CHANG, Y.-C. & SINHA, G. 2007. A visual basic program for ridge axis picking on DEM data using the profile-recognition and polygon-breaking algorithm. *Computers & Geosciences,* 33**,** 229-237.

CHANG, Y.-C., SONG, G.-S. & HSU, S.-K. 1998. Automatic extraction of ridge and valley axes using the profile recognition and polygon-breaking algorithm. *Computers & Geosciences,* 24**,** 83-93.

CHIANG, M.-Y., TU, S.-C., HUANG, J.-Y., TAI, W.-K., LIU, C.-D. & CHANG, C.-C. Terrain synthesis: An interactive approach. International workshop on advanced image tech, 2005.

CHIBA, N., MURAOKA, K. & FUJITA, K. 1998. An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation,* 9**,** 185-194.

COHEN, J. M., HUGHES, J. F. & ZELEZNIK, R. C. 2000. Harold: a world made of drawings. *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering.* Annecy, France: ACM.

COOK, R. L. & DEROSE, T. 2005. Wavelet Noise. *ACM Trans. Graph.,* 24**,** 803-811.

CRIMINISI, A., PÉREZ, P. & TOYAMA, K. 2004. Region filling and object removal by exemplar-based image inpainting. *Image Processing, IEEE Transactions on,* 13**,** 1200-1212.

CUI, J. 2011. *Procedural cave generation.* University of Wollongong.

D'AMBROSIO, D., DI GREGORIO, S., GABRIELE, S. & GAUDIO, R. 2001. A Cellular Automata Model for Soil Erosion by Water. *Physics and Chemistry of the Earth, Part B: Hydrology, Oceans and Atmosphere,* 26**,** 33 - 39.

DACHSBACHER, C. 2006. *Interactive Terrain Rendering: Towards Realism with Procedural Models and Graphics Hardware.* Universität Erlangen–Nürnberg.

DE CARPENTIER, G. J. P. 2007. *Interactively synthesizing and editing virtual outdoor terrain.* Delft University of Technology.

DORSEY, J., EDELMAN, A., JENSEN, H. W., LEGAKIS, J. & PEDERSEN, H. K. Modeling and Rendering of Weathered Stone. Proceedings of the 26th annual conference on Computer graphics and interactive techniques, 1999 New York, NY, USA. ACM Press/Addison-Wesley Publishing Co., 225-234.

DU, P., WEBER, R., LUSZCZEK, P., TOMOV, S., PETERSON, G. & DONGARRA, J. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing,* 38**,** 391-407.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K. & WORLEY, S. 2003. *Texturing and modeling: a procedural approach*, Morgan Kaufmann.

EFROS, A. A. & FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques.* ACM.

EFROS, A. A. & LEUNG, T. K. Texture synthesis by non-parametric sampling. Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on, 1999 1999. 1033-1038 vol.2.

FOURNIER, A., FUSSELL, D. & CARPENTER, L. 1982. Computer Rendering of Stochastic Models. *Commun. ACM,* 25**,** 371-384.

FOWLER, R. J. & LITTLE, J. J. 1979. Automatic Extraction of Irregular Network Digital Terrain Models. *SIGGRAPH Comput. Graph.,* 13**,** 199-207.

GAIN, J., MARAIS, P. & STRAßER, W. Terrain Sketching. Proceedings of the 2009 symposium on Interactive 3D graphics and games, 2009 New York, NY, USA. ACM, 31-38.

GEOGEN. 2013. *Procedural heightmap generator* [Online]. Available: https://code.google.com/p/geogen/.

GEORGE, J. A. 1970. The use of direct methods for the solution of the discrete Poisson equation on non-rectangular regions. Stanford University.

III, T. W. 2015. The Witcher III.

INTEL. 2013. *Xeon Phi* [Online]. Available: www.intel.com/xeonphi.

KARIMI, K., DICKSON, N. G. & HAMZE, F. 2010. A performance comparison of CUDA and OpenCL. *arXiv*.

KAUFMAN, A., COHEN, D. & YAGEL, R. 1993. Volume Graphics. *Computer,* 26**,** 51-64.

KELLEY, A. D., MALIN, M. C. & NIELSON, G. M. 1988. Terrain Simulation Using a Model of Stream Erosion. *SIGGRAPH Comput. Graph.,* 22**,** 263-268.

KHRONOS. 2013. *OpenCL* [Online]. Available: http://www.khronos.org/opencl/.

KRIŠTOF, P., BENEŠ, B., KŘIVÁNEK, J. & ŠT'AVA, O. 2009. Hydraulic Erosion Using Smoothed Particle Hydrodynamics. *Computer Graphics Forum,* 28**,** 219-228.

KRUSKAL, J. B., JR. 1956. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society,* 7**,** 48-50.

KWATRA, V., SCHÖDL, A., ESSA, I., TURK, G. & BOBICK, A. 2003. Graphcut textures: image and video synthesis using graph cuts. *ACM SIGGRAPH 2003 Papers.* San Diego, California: ACM.

LAGAE, A., LEFEBVRE, S., COOK, R., DEROSE, T., DRETTAKIS, G., EBERT, D. S., LEWIS, J. P., PERLIN, K. & ZWICKER, M. 2010. A Survey of Procedural Noise Functions. *Computer Graphics Forum,* 29**,** 2579-2600.

LEWIS, J. P. 1987. Generalized Stochastic Subdivision. *ACM Trans. Graph.,* 6**,** 167-190.

LEWIS, J. P. 1989. Algorithms for Solid Noise Synthesis. *SIGGRAPH Comput. Graph.,* 23**,** 263-270.

LIU, Y., LIN, W.-C. & HAYS, J. 2004. Near-regular texture analysis and manipulation. *ACM Trans. Graph.,* 23**,** 368-376.

LONGMORE, J.-P., MARAIS, P. & KUTTEL, M. M. 2013. Towards realistic and interactive sand simulation: A GPU-based framework. *Powder Technology,* 235**,** 983-1000.

MANDELBROT, B. B. 1975. Stochastic models for the Earth's relief, the shape and the fractal dimension of the coastlines, and the number-area rule for islands. *Proceedings of the National Academy of Sciences,* 72**,** 3825-3828.

MANDELBROT, B. B. 1983. *The Fractal Geometry of Nature*, Times Books.

MANDELBROT, B. B. 1988. The Science of Fractal Images. *In:* PEITGEN, H.-O. & SAUPE, D. (eds.). New York, NY, USA: Springer-Verlag New York, Inc.

MARÁK, I., BENEŠ, B. & SLAVıK, P. Terrain erosion model based on rewriting of matrices. Proceedings of The Fifth International Conference in Central Europe on Computer Graphics and Visualization, 1997. 341-351.

MEI, X., DECAUDIN, P. & HU, B.-G. Fast Hydraulic Erosion Simulation and Visualization on GPU. Computer Graphics and Applications, 2007. PG '07. 15th Pacific Conference on, 2007. 47-56.

MILLER, G. S. P. 1986. The definition and Rendering of Terrain Maps. *SIGGRAPH Comput. Graph.,* 20**,** 39-48.

MILLIRON, T., JENSEN, R. J., BARZEL, R. & FINKELSTEIN, A. 2002. A framework for geometric warps and deformations. *ACM Trans. Graph.,* 21**,** 20-51.

MINECRAFT 2015. Minecraft.

MUSGRAVE, F. K. 1993. Methods for Realistic Landscape Imaging. *Yale University, New Haven, CT*.

MUSGRAVE, F. K., KOLB, C. E. & MACE, R. S. 1989. The Synthesis and Rendering of Eroded Fractal Terrains. *SIGGRAPH Comput. Graph.,* 23**,** 41-50.

NAGASHIMA, K. 1998. Computer generation of eroded valley and mountain terrains. *The Visual Computer,* 13**,** 456-464.

NATALI, M., LIDAL, E. M., PARULEK, J., VIOLA, I. & PATEL, D. Modeling terrains and subsurface geology. Eurographics 2013-State of the Art Reports, 2012. The Eurographics Association, 155-173.

NEIDHOLD, B., WACKER, M. & DEUSSEN, O. 2005. *Interactive physically based Fluid and Erosion Simulation*, Bibliothek der Universität Konstanz.

NIE, D., MA, L. & XIAO, S. Similarity based image inpainting method. Multi-Media Modelling Conference Proceedings, 2006 12th International, 0-0 0 2006. 4 pp.

NVIDIA. 2013a. *CUDA C Best Practices Guide* [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

NVIDIA. 2013b. *CUDA C Programming Guide* [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

NVIDIA. 2013c. *NVIDIA Fermi Compute Architecture Whitepaper* [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

OLSEN, J. 2004. Realtime Procedural Terrain Generation. *Department of Mathematics And Computer Science (IMADA)*.

PAJAROLA, R., ANTONIJUAN, M. & LARIO, R. QuadTIN: Quadtree based Triangulated Irregular Networks. Proceedings of the conference on Visualization '02, 2002 Washington, DC, USA. IEEE Computer Society, 395-402.

PÉREZ, P., GANGNET, M. & BLAKE, A. 2003. Poisson image editing. *ACM SIGGRAPH 2003 Papers.* San Diego, California: ACM.

PERLIN, K. 1985. An Image Synthesizer. *SIGGRAPH Comput. Graph.,* 19**,** 287-296.

PERLIN, K. 2002. Improving Noise. *ACM Trans. Graph.,* 21**,** 681-682.

PEUCKER, T. K., FOWLER, R. J., LITTLE, J. J. & MARK, D. M. The Triangulated Irregular Network. Amer. Soc. Photogrammetry Proc. Digital Terrain Models Symposium, 1978. 532.

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell system technical journal,* 36**,** 1389-1401.

SAUNDERS, R. L. 2006. *Realistic terrain synthesis using genetic algorithms.* Texas A&M University.

SAUPE, D. 1989. Point Evaluation of Multi-Variable Random Fractals. *In:* JÜRGENS, H. & SAUPE, D. (eds.) *Visualisierung in Mathematik und Naturwissenschaften.* Springer Berlin Heidelberg.

SAUPE, D. 1991. Random Fractals in Image Synthesis. *In:* CRILLY, A. J., EARNSHOW, R. A. & JONES, H. (eds.) *Fractals and Chaos.* Springer New York.

SAUPE, D. 2003. Fractals. *Encyclopedia of Computer Science.* Chichester, UK: John Wiley and Sons Ltd.

SCHNEIDER, J., BOLDTE, T. & WESTERMANN, R. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. Vision, modeling and visualization, 2006. 145-152.

SEDGEWICK, R. 2001. *Algorithms in C, Part 5: Graph Algorithms*, Addison-Wesley, Massachusetts.

SHEPARD, D. 1968. A two-dimensional interpolation function for irregularly-spaced data. *Proceedings of the 1968 23rd ACM national conference.* ACM.

SHEWCHUK, J. R. 1994. An introduction to the conjugate gradient method without the agonizing pain. Carnegie-Mellon University. Department of Computer Science.

ŠT'AVA, O., BENEŠ, B., BRISBIN, M. & KŘIVÁNEK, J. Interactive Terrain Modeling Using Hydraulic Erosion. Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2008 Aire-la-Ville, Switzerland, Switzerland. Eurographics Association, 201-210.

STRANDMARK, P. & KAHL, F. Parallel and distributed graph cuts by dual decomposition. Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on, 2010. IEEE, 2085-2092.

TASSE, F. P., EMILIEN, A., CANI, M.-P., HAHMANN, S. & BERNHARDT, A. 2014a. First person sketch-based terrain editing. *Proceedings of Graphics Interface 2014.* Montreal, Quebec, Canada: Canadian Information Processing Society.

TASSE, F. P., EMILIEN, A., CANI, M.-P., HAHMANN, S. & DODGSON, N. 2014b. Feature-based terrain editing from complex sketches. *Computers & Graphics,* 45**,** 101-115.

TASSE, F. P., GAIN, J. & MARAIS, P. 2011. *Distributed Texture-based Terrain Synthesis.* University of Cape Town.

TERRAGEN. 2013. *Terragen 2* [Online]. Available: http://planetside.co.uk/products/terragen2.

THRUST. 2013. *Thrust* [Online]. Available: https://code.google.com/p/thrust/.

USGS. 2013. *US Geological Survey* [Online]. Available: http://nationalmap.gov/.

VOSS, R. F. 1985. Random Fractal Forgeries. *In:* EARNSHAW, R. (ed.) *Fundamental Algorithms for Computer Graphics.* Springer Berlin Heidelberg.

WATANABE, N. & IGARASHI, T. 2004. A sketching interface for terrain modeling. *ACM SIGGRAPH 2004 Posters.* Los Angeles, California: ACM.

WEI, L.-Y. 2003. Texture synthesis from multiple sources. *ACM SIGGRAPH 2003 Sketches & Applications.* San Diego, California: ACM.

WEI, L.-Y., LEFEBVRE, S., KWATRA, V. & TURK, G. State of the Art in Example-based Texture Synthesis. Eurographics 2009, State of the Art Report, EG-STAR, 2009-04-03 2009. Eurographics Association, 93-117.

WORLDMACHINE. 2013. *World Machine 2* [Online]. Available: http://www.world-machine.com/.

ZHOU, H., JIE, S., TURK, G. & REHG, J. M. 2007. Terrain Synthesis from Digital Elevation Models. *Visualization and Computer Graphics, IEEE Transactions on,* 13**,** 834-848.

10 Appendix

10.1 Feature Synthesis – CPU v1 vs. CPU v2

	Small Terrain			Large Terrain		
	Runtime (s)			Runtime (s)		
	CPU v1	CPU v2	Speedup (x)	CPU v1	CPU v2	Speedup (x)
Feature Synthesis	272.100101	270.689996	1.01	5228.739683	5196.345002	1.01
Ridges	192.780592	191.828335	1.00	392.088227	390.610168	1.00
User Patch Generation	0.001063	0.000519	2.05	0.002061	0.001056	1.95
Source Patch Matching	189.037051	188.088532	1.01	382.585721	380.622629	1.01
Source Pre-Processing	0.604155	0.016185	37.33	1.097261	0.016061	68.32
Cost Computation	184.135469	184.382445	1.00	375.573276	374.851056	1.00
Candidate Sorting	1.985055	2.717081	0.73	3.474588	4.724773	0.74
Best Patch Locating	0.000354	0.000618	0.57	0.000707	0.001048	0.67
Patch Merging	3.740209	3.739285	1.00	9.496532	9.986483	0.95
Valleys	79.319514	78.861661	1.01	4836.651464	4805.734834	1.01
User Patch Generation	0.000495	0.000234	2.12	0.028629	0.015006	1.91
Source Patch Matching	77.756361	77.299443	1.01	4760.851660	4730.833789	1.01
Source Pre-Processing	0.294814	0.016550	17.81	15.085132	0.016549	911.57
Cost Computation	74.559710	74.687090	1.00	4668.667237	4663.888232	1.00
Candidate Sorting	0.906126	1.210457	0.75	45.652328	65.524820	0.70
Best Patch Locating	0.000211	0.000253	0.83	0.008847	0.014543	0.61
Patch Merging	1.561903	1.561985	1.00	75.483537	74.886038	1.01

Table 10.1: Runtime results comparing the two main CPU implementations. A speedup column is provided to show the performance gain achieved with version two. These implementations perform very similarly despite the large architectural changes.

10.2 Feature Synthesis – CPU v2 vs. CPU Parallel

	Small Terrain			Large Terrain		
	Runtime (s)			Runtime (s)		
	CPU v2.0	CPU Parallel	Speedup (x)	CPU v2.0	CPU Parallel	Speedup (x)
Feature Synthesis	270.689996	161.831427	1.67	5196.345002	3086.622345	1.68
Ridges	191.828335	114.366444	1.68	390.610168	232.147651	1.68
User Patch Generation	0.000519	0.000517	1.00	0.001056	0.001056	1.00
Source Patch Matching	188.088532	112.166348	1.68	380.622629	226.272194	1.68
Source Pre-Processing	0.016185	0.016177	1.00	0.016061	0.015576	1.03
Cost Computation	184.382445	108.460261	1.70	374.851056	220.500621	1.70
Candidate Sorting	2.717081	2.717063	1.00	4.724773	4.721274	1.00
Best Patch Locating	0.000618	0.000616	1.00	0.001048	0.001046	1.00
Patch Merging	3.739285	2.199579	1.70	9.986483	5.874402	1.70
Valleys	78.861661	47.464983	1.66	4805.734834	2854.474695	1.68
User Patch Generation	0.000234	0.000234	1.00	0.015006	0.014862	1.01
Source Patch Matching	77.299443	46.545935	1.66	4730.833789	2810.409222	1.68
Source Pre-Processing	0.016550	0.016546	1.00	0.016549	0.016547	1.00
Cost Computation	74.687090	43.933582	1.70	4663.888232	2743.463666	1.70
Candidate Sorting	1.210457	1.210437	1.00	65.524820	65.524789	1.00
Best Patch Locating	0.000253	0.000251	1.01	0.014543	0.014518	1.00
Patch Merging	1.561985	0.918814	1.70	74.886038	44.050610	1.70

Table 10.2: Runtime results showing the performance improvements when multithreading our CPU v2 implementation. Only the cost computation stage was multithreaded as such the times for the other sections remain relatively the same.

10.3 Feature Synthesis – CPU Parallel vs. GPU implementations

Small Terrain - Runtime (s)									
	CPU Parallel	GPU v1	GPU v2	GPU v3	GPU v4	GPU v5	GPU v6	GPU v7	GPU v8
Feature Synthesis	161.831427	74.745849	36.122684	36.449545	26.859928	13.793737	14.395617	14.165048	10.541528
Ridges	114.366444	41.478099	20.330847	20.705550	14.756384	8.624020	8.042182	7.874463	6.135917
User Patch Generation	0.000517	0.001940	0.002068	0.002014	0.002007	0.002187	0.001949	0.001894	0.002370
Source Patch Matching	112.166348	39.436534	18.303053	18.629414	12.691242	6.544745	5.993284	5.781124	4.068311
Source Pre-Processing	0.016177	0.006445	0.001121	0.002818	0.004022	0.005167	0.001156	0.001480	0.001870
Cost Computation	108.460261	38.193079	17.075027	17.389231	11.452825	5.299421	4.764113	4.549670	2.835311
Candidate Sorting	2.717063	1.230839	1.224459	1.232209	1.228522	1.233394	1.224980	1.226562	1.226380
Best Patch Locating	0.000616	0.006160	0.002436	0.005144	0.005862	0.006752	0.003026	0.003401	0.004739
Patch Merging	2.199579	2.039625	2.025726	2.074122	2.063136	2.077088	2.046949	2.091445	2.065236
Valleys	47.464983	33.267750	15.791837	15.743996	12.103544	5.169717	6.353435	6.290585	4.405610
User Patch Generation	0.000234	0.001812	0.001983	0.001935	0.002118	0.002045	0.001879	0.001873	0.002110
Source Patch Matching	46.545935	32.420493	14.948588	14.876716	11.240268	4.301775	5.501704	5.420605	3.536686
Source Pre-Processing	0.016546	0.002021	0.001003	0.002328	0.004464	0.006316	0.000937	0.001067	0.002424
Cost Computation	43.933582	31.153961	13.690704	13.612432	9.972212	3.030774	4.243842	4.160566	2.275046
Candidate Sorting	1.210437	1.260102	1.255129	1.259009	1.259777	1.258088	1.255057	1.256038	1.255854
Best Patch Locating	0.000251	0.004389	0.001743	0.002935	0.003804	0.006586	0.001860	0.002924	0.003352
Patch Merging	0.918814	0.845445	0.841266	0.865345	0.861157	0.865897	0.849852	0.868107	0.866815

Large Terrain - Runtime (s)									
	CPU Parallel	GPU v1	GPU v2	GPU v3	GPU v4	GPU v5	GPU v6	GPU v7	GPU v8
Feature Synthesis	3086.622345	859.452684	409.377173	417.778100	316.608775	209.293373	164.387438	164.483185	141.488504
Ridges	232.147651	75.828522	37.523981	38.225733	29.101007	19.279009	17.684657	17.524401	14.672430
User Patch Generation	0.001056	0.061903	0.062274	0.063504	0.063397	0.062755	0.062166	0.062444	0.062705
Source Patch Matching	226.272194	69.002362	30.764750	31.376603	22.216535	12.255220	10.882053	10.420102	7.674574
Source Pre-Processing	0.015576	0.009060	0.004354	0.014392	0.006485	0.011688	0.004435	0.006217	0.007276
Cost Computation	220.500621	66.525701	28.306881	28.891024	19.738832	9.770505	8.419267	7.944341	5.203817
Candidate Sorting	4.721274	2.458855	2.448533	2.463232	2.463144	2.462085	2.452954	2.460839	2.454676
Best Patch Locating	0.001046	0.008722	0.004960	0.007929	0.008050	0.010916	0.005378	0.008682	0.008782
Patch Merging	5.874402	6.764257	6.696957	6.785627	6.821074	6.961035	6.740438	7.041855	6.935151
Valleys	2854.474695	783.624162	371.853192	379.552367	287.507769	190.014363	146.702781	146.958785	126.816074
User Patch Generation	0.014862	0.101599	0.098654	0.098491	0.099536	0.100593	0.097586	0.099739	0.099454
Source Patch Matching	2810.409222	742.197632	330.767996	337.585458	245.507015	148.196523	105.078874	104.975374	84.991130
Source Pre-Processing	0.016547	0.067072	0.049911	0.080541	0.086351	0.099403	0.062882	0.070113	0.073676
Cost Computation	2743.463666	719.303015	308.037628	314.682022	222.585388	125.299297	82.289110	82.113281	62.173043
Candidate Sorting	65.524789	22.739906	22.624760	22.737232	22.747771	22.706748	22.656609	22.696383	22.647466
Best Patch Locating	0.014518	0.087408	0.055452	0.085445	0.087254	0.090821	0.070052	0.095351	0.096694
Patch Merging	44.050610	41.324931	40.986541	41.868418	41.901218	41.717248	41.526321	41.883671	41.725489

Table 10.3: Runtime results comparing the parallel CPU implementation against the different GPU implementations for the small and large terrains. v1 is a translated form of the parallel CPU implementation. v2 adds some shared memory and more threads. v3 attempts to optimise functions but introduces more branching. v4 unrolls an entire loop utilising more concurrent threads. v5 changes the architecture to allow a new dimension of threads for improved concurrency. v6 optimises v5 preventing unnecessary recalculation of values. v7 combines elements from v5 and v6. v8 revisits v4 and incorporates the newer changes in v7.

10.4 Feature Synthesis – Using GPU Texture Memory

Small Terrain						
	Runtime (s)			Speedup over CPU		Speedup over v8
	CPU Parallel	GPU v8	GPU Tex	GPU v8	GPU Tex	
Feature Synthesis	161.831427	10.541528	8.999886	15.35	17.98	1.17
Ridges	114.366444	6.135917	5.373410	18.64	21.28	1.14
User Patch Generation	0.000517	0.002370	0.001987	0.22	0.26	1.19
Source Patch Matching	112.166348	4.068311	3.328250	27.57	33.70	1.22
Source Pre-Processing	0.016177	0.001870	0.001232	8.65	13.13	1.52
Cost Computation	108.460261	2.835311	2.098202	38.25	51.69	1.35
Candidate Sorting	2.717063	1.226380	1.225992	2.22	2.22	1.00
Best Patch Locating	0.000616	0.004739	0.002816	0.13	0.22	1.68
Patch Merging	2.199579	2.065236	2.043173	1.07	1.08	1.01
Valleys	47.464983	4.405610	3.626476	10.77	13.09	1.21
User Patch Generation	0.000234	0.002110	0.002001	0.11	0.12	1.05
Source Patch Matching	46.545935	3.536686	2.761992	13.16	16.85	1.28
Source Pre-Processing	0.016546	0.002424	0.000997	6.83	16.60	2.43
Cost Computation	43.933582	2.275046	1.503504	19.31	29.22	1.51
Candidate Sorting	1.210437	1.255854	1.255539	0.96	0.96	1.00
Best Patch Locating	0.000251	0.003352	0.001945	0.07	0.13	1.72
Patch Merging	0.918814	0.866815	0.862483	1.06	1.07	1.01

Large Terrain						
	Runtime (s)			Speedup over CPU		Speedup over v8
	CPU Parallel	GPU v8	GPU Tex	GPU v8	GPU Tex	
Feature Synthesis	3086.622345	141.488504	125.887884	21.82	24.52	1.12
Ridges	232.147651	14.672430	12.996462	15.82	17.86	1.13
User Patch Generation	0.001056	0.062705	0.067021	0.02	0.02	0.94
Source Patch Matching	226.272194	7.674574	6.220553	29.48	36.37	1.23
Source Pre-Processing	0.015576	0.007276	0.004344	2.14	3.59	1.67
Cost Computation	220.500621	5.203817	3.753462	42.37	58.75	1.39
Candidate Sorting	4.721274	2.454676	2.456292	1.92	1.92	1.00
Best Patch Locating	0.001046	0.008782	0.006433	0.12	0.16	1.37
Patch Merging	5.874402	6.935151	6.708887	0.85	0.88	1.03
Valleys	2854.474695	126.816074	112.891422	22.51	25.29	1.12
User Patch Generation	0.014862	0.099454	0.099500	0.15	0.15	1.00
Source Patch Matching	2810.409222	84.991130	71.666311	33.07	39.22	1.19
Source Pre-Processing	0.016547	0.073676	0.062649	0.22	0.26	1.18
Cost Computation	2743.463666	62.173043	48.853884	44.13	56.16	1.27
Candidate Sorting	65.524789	22.647466	22.666603	2.89	2.89	1.00
Best Patch Locating	0.014518	0.096694	0.082957	0.15	0.18	1.17
Patch Merging	44.050610	41.725489	41.125611	1.06	1.07	1.01

Table 10.4: Runtime results comparing the texture memory GPU implementation compared to the parallel CPU and GPU v8 implementations. There is a slight performance gain when using texture memory. This is because we already are using coalesced memory access for our image data. The first two speedup columns are comparing the methods against the CPU implementation with the last speedup value comparing the improvement texture memory provides compared to the current best GPU v8 implementation.

10.5 Feature Synthesis – CPU vs. GPU Candidate Sorting

Small Terrain						
	Runtime (s)			Speedup over CPU		Speedup over GPU
	CPU Sorting	GPU Sorting	Thrust Sorting	GPU Sorting	Thrust Sorting	
Feature Synthesis	12.678237	9.022137	7.037827	1.41	1.80	1.28
Ridges	8.356787	5.398602	4.511774	1.55	1.85	1.20
User Patch Generation	0.002545	0.002066	0.002370	1.23	1.07	0.87
Source Patch Matching	6.283441	3.329156	2.420604	1.89	2.60	1.38
Source Pre-Processing	0.003036	0.001437	0.001084	2.11	2.80	1.33
Cost Computation	2.109137	2.098663	2.107912	1.00	1.00	1.00
Candidate Sorting	4.141839	1.225630	0.300835	3.38	13.77	4.07
Best Patch Locating	0.007287	0.003416	0.004354	2.13	1.67	0.78
Patch Merging	2.070801	2.067380	2.088801	1.00	0.99	0.99
Valleys	4.321450	3.623535	2.526053	1.19	1.71	1.43
User Patch Generation	0.002175	0.002236	0.002044	0.97	1.06	1.09
Source Patch Matching	3.453277	2.761616	1.663156	1.25	2.08	1.66
Source Pre-Processing	0.001839	0.001726	0.000757	1.07	2.43	2.28
Cost Computation	1.500838	1.502602	1.505681	1.00	1.00	1.00
Candidate Sorting	1.933492	1.255184	0.151018	1.54	12.80	8.31
Best Patch Locating	0.005162	0.002096	0.003278	2.46	1.57	0.64
Patch Merging	0.865998	0.859684	0.860853	1.01	1.01	1.00

Large Terrain						
	Runtime (s)			Speedup over CPU		Speedup over GPU
	CPU Sorting	GPU Sorting	Thrust Sorting	GPU Sorting	Thrust Sorting	
Feature Synthesis	208.050630	126.973870	111.550104	1.64	1.87	1.14
Ridges	18.351054	13.278647	11.529683	1.38	1.59	1.15
User Patch Generation	0.066535	0.066883	0.068119	0.99	0.98	0.98
Source Patch Matching	11.267451	6.258385	4.471157	1.80	2.52	1.40
Source Pre-Processing	0.007293	0.005249	0.003516	1.39	2.07	1.49
Cost Computation	3.835242	3.790283	3.762511	1.01	1.02	1.01
Candidate Sorting	7.373658	2.455467	0.684837	3.00	10.77	3.59
Best Patch Locating	0.015873	0.007365	0.011246	2.16	1.41	0.65
Patch Merging	7.017067	6.953378	6.990407	1.01	1.00	0.99
Valleys	189.699576	113.695224	100.020421	1.67	1.90	1.14
User Patch Generation	0.112380	0.104432	0.102666	1.08	1.09	1.02
Source Patch Matching	147.562560	71.802278	58.209333	2.06	2.54	1.23
Source Pre-Processing	0.081786	0.070201	0.043295	1.17	1.89	1.62
Cost Computation	49.481234	49.000015	48.833930	1.01	1.01	1.00
Candidate Sorting	97.366113	22.651851	9.108347	4.30	10.69	2.49
Best Patch Locating	0.179276	0.079965	0.108337	2.24	1.65	0.74
Patch Merging	42.024636	41.788514	41.708422	1.01	1.01	1.00

Table 10.5: Runtime results comparing sorting of the candidates with the CPU, our own GPU kernel or using the Thrust (2013) library. We observe a large speedup when using the GPU to sort candidates, which is further increased when using the optimised Thrust library. The first two speedup columns compare the GPU sorting algorithms to CPU sorting with the final speedup value comparing the improvement Thrust provides over our implementation.

10.6 Feature Synthesis – Asynchronous Blocked Implementation

Small Terrain	Runtime (s)			Speedup over CPU		Speedup over Thrust
	CPU Parallel	GPU Thrust	GPU Async	GPU Thrust	GPU Async	
Feature Synthesis	161.831427	7.037827	6.607751	22.99	24.49	1.07
Ridges	114.366444	4.511774	4.134849	25.35	27.66	1.09
User Patch Generation	0.000517	0.002370	0.001885	0.22	0.27	1.26
Source Patch Matching	112.166348	2.420604	2.375004	46.34	47.23	1.02
Source Pre-Processing	0.016177	0.001084	0.000795	14.92	20.35	1.36
Cost Computation	108.460261	2.107912	2.067151	51.45	52.47	1.02
Candidate Sorting	2.717063	0.300835	0.297279	9.03	9.14	1.01
Best Patch Locating	0.000616	0.004354	0.003483	0.14	0.18	1.25
Patch Merging	2.199579	2.088801	1.755116	1.05	1.25	1.19
Valleys	47.464983	2.526053	2.472902	18.79	19.19	1.02
User Patch Generation	0.000234	0.002044	0.001798	0.11	0.13	1.14
Source Patch Matching	46.545935	1.663156	1.626961	27.99	28.61	1.02
Source Pre-Processing	0.016546	0.000757	0.000749	21.87	22.09	1.01
Cost Computation	43.933582	1.505681	1.479801	29.18	29.69	1.02
Candidate Sorting	1.210437	0.151018	0.141718	8.02	8.54	1.07
Best Patch Locating	0.000251	0.003278	0.002319	0.08	0.11	1.41
Patch Merging	0.918814	0.860853	0.842810	1.07	1.09	1.02

Large Terrain	Runtime (s)			Speedup over CPU		Speedup over Thrust
	CPU Parallel	GPU Thrust	GPU Async	GPU Thrust	GPU Async	
Feature Synthesis	3086.622345	111.550104	68.531263	27.67	45.04	1.63
Ridges	232.147651	11.529683	10.130781	20.13	22.92	1.14
User Patch Generation	0.001056	0.068119	0.065550	0.02	0.02	1.04
Source Patch Matching	226.272194	4.471157	4.358404	50.61	51.92	1.03
Source Pre-Processing	0.015576	0.003516	0.003343	4.43	4.66	1.05
Cost Computation	220.500621	3.762511	3.702657	58.60	59.55	1.02
Candidate Sorting	4.721274	0.684837	0.635609	6.89	7.43	1.08
Best Patch Locating	0.001046	0.011246	0.007806	0.09	0.13	1.44
Patch Merging	5.874402	6.990407	6.950981	0.84	0.85	1.01
Valleys	2854.474695	100.020421	58.400482	28.54	48.88	1.71
User Patch Generation	0.014862	0.102666	0.096177	0.14	0.15	1.07
Source Patch Matching	2810.409222	58.209333	55.846058	48.28	50.32	1.04
Source Pre-Processing	0.016547	0.043295	0.026975	0.38	0.61	1.61
Cost Computation	2743.463666	48.833930	47.873112	56.18	57.31	1.02
Candidate Sorting	65.524789	9.108347	7.732866	7.19	8.47	1.18
Best Patch Locating	0.014518	0.108337	0.089898	0.13	0.16	1.21
Patch Merging	44.050610	41.708422	42.241699	1.06	1.04	0.99

Table 10.6: Runtime results comparing the parallel CPU and our current best GPU implementation, using Thrust sorting, against our asynchronous block system. This allows us to execute code on both the CPU and GPU concurrently, which produces a very large improvement over our current best GPU implementation. The first two speedup columns are compared to our parallel CPU implementation with the last indicating the gain when using asynchronous processing over the Thrust enabled GPU implementation.

10.7 Feature Synthesis – Culling Nearby User Patches

	Small Terrain			Large Terrain		
	Runtime (s)			Runtime (s)		
	Culling Off	Culling On	Speedup (x)	Culling Off	Culling On	Speedup (x)
Feature Synthesis	44.575818	22.717946	1.96	210.679105	176.090025	1.20
Ridges	26.311706	11.340142	2.32	115.222443	92.418828	1.25
User Patch Generation	0.008714	0.004231	2.06	0.105202	0.101863	1.03
Source Patch Matching	17.702781	6.990403	2.53	70.422854	55.509404	1.27
Source Pre-Processing	0.014339	0.007372	1.94	0.120667	0.075329	1.60
Cost Computation	11.549297	4.516402	2.56	46.792403	36.916863	1.27
Candidate Sorting	6.115225	2.456392	2.49	23.422729	18.438620	1.27
Best Patch Locating	0.023861	0.010214	2.34	0.086826	0.078439	1.11
Patch Merging	8.600211	4.345508	1.98	44.694387	36.807561	1.21
Valleys	18.264112	11.377804	1.61	95.456662	83.671197	1.14
User Patch Generation	0.005557	0.004366	1.27	0.094586	0.094699	1.00
Source Patch Matching	11.849165	7.214704	1.64	60.995559	53.084093	1.15
Source Pre-Processing	0.009069	0.005512	1.65	0.127965	0.066334	1.93
Cost Computation	8.041921	4.675941	1.72	40.581834	35.303080	1.15
Candidate Sorting	3.785017	2.523086	1.50	20.213967	17.643906	1.15
Best Patch Locating	0.013124	0.010145	1.29	0.071608	0.070622	1.01
Patch Merging	6.409391	4.158733	1.54	34.366517	30.492405	1.13

Table 10.7: Runtime results comparing the implementations when either culling of nearby user patches or not. This is an issue with the original feature extraction algorithm. We address this by examining user patches and removing those that are in close proximity to one another. This reduces the total number of features requiring synthesis and thus improves performance as shown above. We see a higher gain in the smaller terrain as the proportion of culled patches is higher than the larger terrain.

10.8 Feature Synthesis – Feature Complexity Change

Runtime (s)				
	Feature Count			
	380	770	1145	1541
Feature Synthesis	30.939944	53.472603	78.266373	110.815706
Ridges	13.707008	19.945270	26.986280	38.879797
User Patch Generation	0.067431	0.074991	0.076889	0.085137
Source Patch Matching	8.630530	17.373795	25.460282	37.438202
Source Pre-Processing	0.004646	0.011593	0.022171	0.043379
Cost Computation	7.492320	14.761641	20.742458	28.729611
Candidate Sorting	1.104025	2.526881	4.588528	8.499628
Best Patch Locating	0.013307	0.035492	0.052619	0.088187
Patch Merging	10.138998	16.375170	21.924746	29.305117
Valleys	17.232936	33.527334	51.280093	71.935909
User Patch Generation	0.070370	0.081682	0.087379	0.103568
Source Patch Matching	15.593666	31.254787	49.137721	63.922107
Source Pre-Processing	0.007824	0.018771	0.039102	0.074120
Cost Computation	13.435021	26.261687	39.759585	53.003480
Candidate Sorting	2.093096	4.850851	9.136727	16.552344
Best Patch Locating	0.024511	0.054753	0.094800	0.151200
Patch Merging	10.768992	22.829994	34.641575	47.037576

Table 10.8: Runtime results for varying complexity in terms of the number of total features synthesised by the system. We observe that with a linear increase in the total number of features there is a linear increase in the time required. This allows our system to scale for larger more complex terrains.

10.9 Non-Feature Synthesis

	Small Terrain			Large Terrain		
	CPU	GPU Enhanced	Speedup (x)	CPU	GPU Enhanced	Speedup (x)
Non-Feature Synthesis	25.657241	7.261747	3.53	5574.593558	3270.905490	1.70
Generate Source Candidates	0.158949	0.015000	10.60	0.158722	0.014966	10.61
Initialise Omega	0.065268	0.064307	1.01	1.471440	1.466030	1.00
Get Next Target	0.442839	0.443905	1.00	1840.985378	1902.049310	0.97
Sorting Omega	0.064358	0.065023	0.99	529.135244	543.928055	0.97
Cost Calculation	19.872205	0.096544	205.84	2604.309317	14.880738	175.01
Sorting Candidates	0.058681	1.535284	0.04	5.261691	217.122643	0.02
Find Best Patch	0.405882	0.455423	0.89	53.408648	51.875471	1.03
Merging Patch	4.589059	4.586262	1.00	539.863117	539.568277	1.00

Table 10.9: Runtime results for the non-feature synthesis stage of our system. Times presented are for a CPU only and GPU enhanced implementations. The GPU is utilised for cost calculations to help reduce the overhead of synthesis, the other components are left CPU bound. There is a massive improvement in the cost calculation stage, which has the largest runtime on the CPU.

10.10 Full Synthesis – Previous Work

Small Terrain							
	Runtime (s)				Speedup over Tasse et al.		
	Tasse et al.	CPU v2.0	CPU Parallel	GPU Best	CPU v2.0	CPU Parallel	GPU Best
Total Synthesis	567.474000	13.140256	10.936587	8.203077	43.19	51.89	69.18
Feature Synthesis	201.712000	5.685627	3.400151	2.378101	35.48	59.32	84.82
Ridges	140.849000	3.851331	2.296131	1.681765	36.57	61.34	83.75
Valleys	60.863000	1.834296	1.104020	0.696336	33.18	55.13	87.40
Non-Feature Synthesis	365.762000	7.454629	7.536437	5.824976	49.07	48.53	62.79

Large Terrain							
	Runtime (s)				Speedup over Tasse et al.		
	Tasse et al.	CPU v2.0	CPU Parallel	GPU Best	CPU v2.0	CPU Parallel	GPU Best
Total Synthesis	7460.135000	226.967237	216.476559	176.280177	32.87	34.46	42.32
Feature Synthesis	654.425000	28.668165	17.028104	9.394366	22.83	38.43	69.66
Ridges	654.425000	28.668165	17.028104	9.394366	22.83	38.43	69.66
Non-Feature Synthesis	6805.710000	198.299072	199.448455	166.885811	34.32	34.12	40.78

Table 10.10: Runtime results when comparing our system to the previous work by Tasse et al. (2011). Timing values for Ridges, Valleys and Non-Feature Synthesis were provided in the previous system as such we omit the breakdown for our system in order to only compare the relevant data. While we could only compare the CPU implementation of Tasse et al. (2011), we observe that our system runs significantly faster under the same test conditions. Our system was run with a single source file to match the output more closely.

10.11 Full Synthesis – Single vs. Multiple Sources

	Small Terrain			Large Terrain		
	Runtime (s)			Runtime (s)		
	Single Source	Multi-Source	Speedup (x)	Single Source	Multi-Source	Speedup (x)
Total Synthesis	7.819481	15.405385	0.51	3049.398856	3217.503113	0.95
Feature Synthesis	2.094224	8.662770	0.24	35.790310	82.477729	0.43
Ridges	1.391658	5.291615	0.26	6.244918	10.848514	0.58
User Patch Generation	0.001920	0.002221	0.86	0.063670	0.060909	1.05
Source Patch Matching	0.058827	3.224222	0.02	0.133099	5.796704	0.02
Source Pre-Processing	0.000129	0.001084	0.12	0.000441	0.003274	0.13
Cost Computation	0.038935	2.899855	0.01	0.074109	5.134853	0.01
Candidate Sorting	0.018687	0.309995	0.06	0.055980	0.643227	0.09
Best Patch Locating	0.000803	0.004745	0.17	0.001960	0.006830	0.29
Patch Merging	1.328188	2.062456	0.64	6.088430	6.950793	0.88
Valleys	0.702566	3.371155	0.21	29.545392	71.629215	0.41
User Patch Generation	0.001866	0.001657	1.13	0.094067	0.093394	1.01
Source Patch Matching	0.047725	2.486569	0.02	1.521386	69.111122	0.02
Source Pre-Processing	0.000004	0.000685	0.01	0.004004	0.031189	0.13
Cost Computation	0.037896	2.330338	0.02	0.957425	60.912516	0.02
Candidate Sorting	0.009165	0.149637	0.06	0.538805	7.964012	0.07
Best Patch Locating	0.000480	0.003563	0.13	0.014083	0.086265	0.16
Patch Merging	0.651718	0.881730	0.74	29.161845	41.278248	0.71
Non-Feature Synthesis	5.725257	6.742615	0.85	3013.608546	3135.025384	0.96
Generate Source Candidates	0.003451	0.030741	0.11	0.003091	0.029903	0.10
Initialise Omega	0.074822	0.065653	1.14	1.662981	1.463157	1.14
Get Next Target	0.445233	0.452896	0.98	1836.908801	1863.952490	0.99
Sorting Omega	0.059441	0.066826	0.89	532.757324	542.432483	0.98
Cost Calculation	0.102188	0.100574	1.02	14.885734	14.737024	1.01
Sorting Candidates	0.144254	0.907089	0.16	27.213139	122.224458	0.22
Find Best Patch	0.391972	0.447185	0.88	49.892418	51.130308	0.98
Merging Patch	4.503896	4.671652	0.96	550.285059	539.055560	1.02

Table 10.11: Runtime results for our system when using either a single input source or our database of fifteen. We see the feature synthesis stage has a fairly high cost for using multiple files, although less so when using the larger terrain. We observe the runtimes for non-feature synthesis being very close between the two implementations due to the large cost of running many iterations to completely fill the output terrain. When looking at the total synthesis time for the large terrain we see the larger database has very minor impact on the performance.

10.12 Full Synthesis – Varying Patch Size

Small Terrain	Runtime (s)					Speedup over 32px			
	32 x 32	64 x 64	96 x 96	128 x 128	160 x 160	64 x 64	96 x 96	128 x 128	160 x 160
Total Synthesis	17.702534	13.980580	16.004881	18.995432	22.861861	1.27	1.11	0.93	0.77
Feature Synthesis	7.792151	7.060216	8.685814	11.260066	15.848494	1.10	0.90	0.69	0.49
Ridges	5.281415	4.508658	5.209432	6.822661	9.551658	1.17	1.01	0.77	0.55
User Patch Generation	0.002206	0.002190	0.002568	0.002733	0.002768	1.01	0.86	0.81	0.80
Source Patch Matching	4.909848	2.432063	1.945530	1.989511	2.091009	2.02	2.52	2.47	2.35
Source Pre-Processing	0.005797	0.001120	0.001133	0.002560	0.001881	5.17	5.12	2.26	3.08
Cost Computation	4.236849	2.118850	1.760964	1.855808	1.989788	2.00	2.41	2.28	2.13
Candidate Sorting	0.646004	0.301765	0.177201	0.126182	0.094104	2.14	3.65	5.12	6.86
Best Patch Locating	0.010367	0.004143	0.003345	0.002703	0.004112	2.50	3.10	3.84	2.52
Patch Merging	1.031753	2.074405	3.261334	4.830417	7.457881	0.50	0.32	0.21	0.14
Valleys	2.510737	2.551558	3.476383	4.437405	6.296836	0.98	0.72	0.57	0.40
User Patch Generation	0.001535	0.002135	0.002297	0.002205	0.002518	0.72	0.67	0.70	0.61
Source Patch Matching	2.212971	1.688215	1.588943	1.624058	1.875935	1.31	1.39	1.36	1.18
Source Pre-Processing	0.001910	0.000880	0.000835	0.000767	0.001159	2.17	2.29	2.49	1.65
Cost Computation	1.895923	1.529103	1.485853	1.557852	1.808238	1.24	1.28	1.22	1.05
Candidate Sorting	0.305725	0.152603	0.097979	0.062149	0.062734	2.00	3.12	4.92	4.87
Best Patch Locating	0.004918	0.003257	0.002662	0.002041	0.002991	1.51	1.85	2.41	1.64
Patch Merging	0.295131	0.861208	1.885143	2.811143	4.418384	0.34	0.16	0.10	0.07
Non-Feature Synthesis	9.910383	6.920364	7.319066	7.735366	7.013367	1.43	1.35	1.28	1.41
Generate Source Candidates	0.000004	0.000004	0.000004	0.000004	0.000004	1.00	1.00	1.00	1.00
Initialise Omega	0.021156	0.064253	0.093208	0.145327	0.135087	0.33	0.23	0.15	0.16
Get Next Target	0.902328	0.462035	0.275398	0.187707	0.126047	1.95	3.28	4.81	7.16
Sorting Omega	0.434196	0.067906	0.014281	0.004862	0.001345	6.39	30.40	89.31	322.88
Cost Calculation	0.121113	0.097208	0.079778	0.030024	0.022988	1.25	1.52	4.03	5.27
Sorting Candidates	4.630013	1.116152	0.446110	0.090106	0.043091	4.15	10.38	51.38	107.45
Find Best Patch	0.980026	0.487460	0.381753	0.387719	0.288513	2.01	2.57	2.53	3.40
Merging Patch	2.817235	4.609964	5.993800	6.858287	6.350219	0.61	0.47	0.41	0.44

Large Terrain	Runtime (s)					Speedup over 32px			
	32 x 32	64 x 64	96 x 96	128 x 128	160 x 160	64 x 64	96 x 96	128 x 128	160 x 160
Total Synthesis	9499.526314	3298.099709	2365.878455	2416.259514	2380.718240	2.88	4.02	3.93	3.99
Feature Synthesis	114.020946	70.388126	98.469137	165.028469	221.776193	1.62	1.16	0.69	0.51
Ridges	7.955031	10.264930	12.858465	18.108056	21.434686	0.77	0.62	0.44	0.37
User Patch Generation	0.067114	0.068951	0.068756	0.071802	0.067217	0.97	0.98	0.93	1.00
Source Patch Matching	6.955356	4.486431	2.936234	3.051411	2.417770	1.55	2.37	2.28	2.88
Source Pre-Processing	0.006750	0.003672	0.002336	0.002753	0.002576	1.84	2.89	2.45	2.62
Cost Computation	5.666742	3.777195	2.506313	2.729738	2.202575	1.50	2.26	2.08	2.57
Candidate Sorting	1.246707	0.685703	0.417435	0.309137	0.206566	1.82	2.99	4.03	6.04
Best Patch Locating	0.017734	0.010848	0.004354	0.004977	0.003867	1.63	4.07	3.56	4.59
Patch Merging	5.207338	7.000171	9.848699	14.979439	18.943920	0.74	0.53	0.35	0.27
Valleys	106.065915	60.123197	85.610673	146.920413	200.341507	1.76	1.24	0.72	0.53
User Patch Generation	0.107495	0.096717	0.095620	0.098486	0.096554	1.11	1.12	1.09	1.11
Source Patch Matching	105.327639	57.551077	39.433866	32.680596	27.176944	1.83	2.67	3.22	3.88
Source Pre-Processing	0.061594	0.033146	0.018780	0.020914	0.015399	1.86	3.28	2.95	4.00
Cost Computation	88.098688	48.711506	34.357757	28.930040	24.549764	1.81	2.56	3.05	3.59
Candidate Sorting	16.695155	8.568948	4.921386	3.598347	2.534429	1.95	3.39	4.64	6.59
Best Patch Locating	0.229707	0.110273	0.056810	0.065475	0.045073	2.08	4.04	3.51	5.10
Patch Merging	16.794090	42.244356	82.069947	142.918269	196.170912	0.40	0.20	0.12	0.09
Non-Feature Synthesis	9385.505368	3227.711583	2267.409318	2251.231045	2158.942047	2.91	4.14	4.17	4.35
Generate Source Candidates	0.000004	0.000004	0.000004	0.000004	0.000004	1.00	1.00	1.00	1.00
Initialise Omega	0.452514	1.473950	3.576196	6.005665	9.253114	0.31	0.13	0.08	0.05
Get Next Target	5255.923706	1865.092529	1103.501844	926.461208	746.169029	2.82	4.76	5.67	7.04
Sorting Omega	3118.877476	539.060791	203.509992	121.919444	78.433835	5.79	15.33	25.58	39.76
Cost Calculation	16.200290	14.900963	14.101240	6.087718	5.784305	1.09	1.15	2.66	2.80
Sorting Candidates	591.789447	216.331482	87.426378	17.464585	12.029516	2.74	6.77	33.89	49.19
Find Best Patch	100.179838	52.393124	53.907717	60.806208	63.783032	1.91	1.86	1.65	1.57
Merging Patch	302.082093	538.458740	801.385946	1112.486212	1243.489212	0.56	0.38	0.27	0.24

Table 10.12: Runtime results for varying the size of the patch used by our system. We start off with a small 32×32 patch size up to a large 160×160 patch size. We observe two outcomes when looking at the feature and non-feature synthesis components, which is similar for both terrain sizes. For feature synthesis we see a patch size of 64×64 being optimal with the fastest runtime recorded. For non-feature synthesis we observe that the larger the patch size the faster the runtime. This is attributed to a larger area being merged into the output, which reduces the amount of empty areas thus requiring less iterations to complete.